高质量 C++/C 编程指南

<table>
<tr><td rowspan="4">文件状态
[ ] 草稿文件
[√] 正式文件
[ ] 更改正式文件</td><td>文件标识：</td><td></td></tr>
<tr><td>当前版本：</td><td>1.0</td></tr>
<tr><td>作　　者：</td><td>林锐 博士</td></tr>
<tr><td>完成日期：</td><td>2001 年 7 月 24 日</td></tr>
</table>

版 本 历 史

版本/状态	作者	参与者	起止日期	备注
V 0.9 草稿文件	林锐		**2001-7-1** 至 **2001-7-18**	林锐起草
V 1.0 正式文件	林锐		**2001-7-18** 至 **2001-7-24**	朱洪海审查 **V 0.9**， 林锐修正草稿中的错误

目 录

前 言

软件质量是被大多数程序员挂在嘴上而不是放在心上的东西！

除了完全外行和真正的编程高手外，初读本书，你最先的感受将是惊慌：“哇！我以前捏造的 C++/C 程序怎么会有那么多的毛病？”

别难过，作者只不过比你早几年、多几次惊慌而已。

请花一两个小时认真阅读这本百页经书，你将会获益匪浅，这是前面 N-1 个读者的建议。

一、编程老手与高手的误区

自从计算机问世以来，程序设计就成了令人羡慕的职业，程序员在受人宠爱之后容易发展成为毛病特多却常能自我臭美的群体。

如今在 Internet 上流传的“真正”的程序员据说是这样的：

(1) 真正的程序员没有进度表，只有讨好领导的马屁精才有进度表，真正的程序员会让领导提心吊胆。
(2) 真正的程序员不写使用说明书，用户应当自己去猜想程序的功能。
(3) 真正的程序员几乎不写代码的注释，如果注释很难写，它理所当然也很难读。
(4) 真正的程序员不画流程图，原始人和文盲才会干这事。
(5) 真正的程序员不看参考手册，新手和胆小鬼才会看。
(6) 真正的程序员不写文档也不需要文档，只有看不懂程序的笨蛋才用文档。
(7) 真正的程序员认为自己比用户更明白用户需要什么。
(8) 真正的程序员不接受团队开发的理念，除非他自己是头头。
(9) 真正的程序员的程序不会在第一次就正确运行，但是他们愿意守着机器进行若干个 30 小时的调试改错。
(10) 真正的程序员不会在上午 9:00 到下午 5:00 之间工作，如果你看到他在上午 9:00 工作，这表明他从昨晚一直干到现在。

……

具备上述特征越多，越显得水平高，资格老。所以别奇怪，程序员的很多缺点竟然可以被当作优点来欣赏。就象在武侠小说中，那些独来独往、不受约束且带点邪气的高手最令人崇拜。我曾经也这样信奉，并且希望自己成为那样的“真正”的程序员，结果没有得到好下场。

我从读大学到博士毕业十年来一直勤奋好学，累计编写了数十万行 C++/C 代码。有这样的苦劳和疲劳，我应该称得上是编程老手了吧？

我开发的软件都与科研相关（集成电路 CAD 和 3D 图形学领域），动辄数万行程序，技术复杂，难度颇高。这些软件频频获奖，有一个软件获得首届中国大学生电脑大赛软

件展示一等奖。在 1995 年开发的一套图形软件库到 2000 年还有人买。罗列出这些“业绩”，可以说明我算得上是编程高手了吧？

可惜这种个人感觉不等于事实。

读博期间我曾用一年时间开发了一个近 10 万行 C++代码的 3D 图形软件产品，我内心得意表面谦虚地向一位真正的软件高手请教。他虽然从未涉足过 3D 图形领域，却在几十分钟内指出该软件多处重大设计错误。让人感觉那套软件是用纸糊的华丽衣服，扯一下掉一块，戳一下破个洞。我目瞪口呆地意识到这套软件毫无实用价值，一年的心血白化了，并且害死了自己的软件公司。

人的顿悟通常发生在最心痛的时刻，在沮丧和心痛之后，我作了深刻反省，“面壁”半年，重新温习软件设计的基础知识。补修“内功”之后，又觉得腰板硬了起来。博士毕业前半年，我曾到微软中国研究院找工作，接受微软公司一位资深软件工程师的面试。他让我写函数 strcpy 的代码。

太容易了吧？

错！

这么一个小不点的函数，他从三个方面考查：

（1）编程风格；

（2）出错处理；

（3）算法复杂度分析（用于提高性能）。

在大学里从来没有人如此严格地考查过我的程序。我花了半个小时，修改了数次，他还不尽满意，让我回家好好琢磨。我精神抖擞地进“考场”，大汗淋漓地出“考场”。这“高手”当得也太窝囊了。我又好好地反省了一次。

我把反省后的心得体会写成文章放在网上传阅，引起了不少软件开发人员的共鸣。我因此有幸和国产大型 IT 企业如华为、上海贝尔、中兴等公司的同志们广泛交流。大家认为提高质量与生产率是软件工程要解决的核心问题。高质量程序设计是非常重要的环节，毕竟软件是靠编程来实现的。

我们心目中的老手们和高手们能否编写出高质量的程序来？

不见得都能！

就我的经历与阅历来看，国内大学的计算机教育压根就没有灌输高质量程序设计的观念，教师们和学生们也很少自觉关心软件的质量。勤奋好学的程序员长期在低质量的程序堆中滚爬，吃尽苦头之后才有一些心得体会，长进极慢，我就是一例。

现在国内 IT 企业拥有学士、硕士、博士文凭的软件开发人员比比皆是，但他们在接受大学教育时就“先天不足”，岂能一到企业就突然实现质的飞跃。试问有多少软件开发人员对正确性、健壮性、可靠性、效率、易用性、可读性（可理解性）、可扩展性、可复用性、兼容性、可移植性等质量属性了如指掌？并且能在实践中运用自如？。“高质量”可不是干活小心点就能实现的！

我们有充分的理由疑虑：

（1）编程老手可能会长期用隐含错误的方式编程（习惯成自然），发现毛病后都不愿相信那是真的！

（2）编程高手可以在某一领域写出极有水平的代码，但未必能从全局把握软件质量的方方面面。

事实证明如此。我到上海贝尔工作一年来，陆续面试或测试过近百名“新”“老”程序员的编程技能，质量合格率大约是 10%。很少有人能够写出完全符合质量要求的 if 语句，很多程序员对指针、内存管理一知半解，……。

领导们不敢相信这是真的。我做过现场试验：有一次部门新进 14 名硕士生，在开欢迎会之前对他们进行“C++/C 编程技能”摸底考试。我问大家试题难不难？所有的人都回答不难。结果没有一个人及格，有半数人得零分。竞争对手公司的朋友们也做过试验，同样一败涂地。

真的不是我“心狠手辣”或者要求过高，而是很多软件开发人员对自己的要求不够高。

要知道华为、上海贝尔、中兴等公司的员工素质在国内 IT 企业中是比较前列的，倘若他们的编程质量都如此差的话，我们怎么敢期望中小公司拿出高质量的软件呢？连程序都编不好，还谈什么振兴民族软件产业，岂不胡扯。

我打算定义编程老手和编程高手，请您别见笑。

定义 1：能长期稳定地编写出高质量程序的程序员称为编程老手。

定义 2：能长期稳定地编写出高难度、高质量程序的程序员称为编程高手。

根据上述定义，马上得到第一推论：我既不是高手也算不上是老手。

在写此书前，我阅读了不少程序设计方面的英文著作，越看越羞惭。因为发现自己连编程基本技能都未能全面掌握，顶多算是二流水平，还好意思谈什么老手和高手。希望和我一样在国内土生土长的程序员朋友们能够做到：

（1）知错就改；

（2）经常温故而知新；

（3）坚持学习，天天向上。

二、本书导读

首先请做附录 B 的 C++/C 试题（不要看答案），考查自己的编程质量究竟如何。然后参照答案严格打分。

（1）如果你只得了几十分，请不要声张，也不要太难过。编程质量差往往是由于不良习惯造成的，与人的智力、能力没有多大关系，还是有药可救的。成绩越差，可以进步的空间就越大，中国不就是在落后中赶超发达资本主义国家吗？只要你能下决心改掉不良的编程习惯，第二次考试就能及格了。

（2）如果你考及格了，表明你的技术基础不错，希望你能虚心学习、不断进步。如果你还没有找到合适的工作单位，不妨到上海贝尔试一试。

（3）如果你考出 85 分以上的好成绩，你有义务和资格为你所在的团队作“ C++/C 编程 ”培训。希望你能和我们多多交流、相互促进。半年前我曾经发现一颗好苗子，就把他挖到我们小组来。

（4）如果你在没有任何提示的情况下考了满分，希望你能收我做你的徒弟。

编程考试结束后，请阅读本书的正文。

本书第一章至第六章主要论述 C++/C 编程风格。难度不高，但是细节比较多。别小看了，提高质量就是要从这些点点滴滴做起。世上不存在最好的编程风格，一切因需求而定。团队开发讲究风格一致，如果制定了大家认可的编程风格，那么所有组员都要遵守。如果读者觉得本书的编程风格比较合你的工作，那么就采用它，不要只看不做。人在小时候说话发音不准，写字潦草，如果不改正，总有后悔的时候。编程也是同样道理。

第七章至第十一章是专题论述，技术难度比较高，看书时要积极思考。特别是第七章“ 内存管理 ”，读了并不表示懂了，懂了并不表示就能正确使用。有一位同事看了第七章后觉得“ 野指针 ”写得不错，与我切磋了一把。可是过了两周，他告诉我，他忙了两天追查出一个 Bug，想不到又是“ 野指针 ”出问题，只好重读第七章。

光看本书对提高编程质量是有限的，建议大家阅读本书的参考文献，那些都是经典名著。

如果你的编程质量已经过关了，不要就此满足。如果你想成为优秀的软件开发人员，建议你阅读并按照 **CMMI** 规范做事，让自己的综合水平上升一个台阶。上海贝尔的员工可以向网络应用事业部软件工程研究小组索取 **CMMI** 有关资料，最好能参加培训。

三、版权声明

本书的大部分内容取材于作者一年前的书籍手稿（尚未出版），现整理汇编成为上海贝尔网络应用事业部的一个规范化文件，同时作为培训教材。

由于 C++/C 编程是众所周知的技术，没有秘密可言。编程的好经验应该大家共享，我们自己也是这么学来的。作者愿意公开本书的电子文档。

版权声明如下：

（1）读者可以任意拷贝、修改本书的内容，但不可以篡改作者及所属单位。

（2）未经作者许可，不得出版或大量印发本书。

（3）如果竞争对手公司的员工得到本书，请勿公开使用，以免发生纠纷。

预计到 2002 年 7 月，我们将建立切合中国国情的 CMMI 3 级解决方案。届时，包括本书在内的约 1000 页规范将严格受控。

欢迎读者对本书提出批评建议。

林锐，2001 年 7 月

第 1 章 文件结构

每个 **C++/C** 程序通常分为两个文件。一个文件用于保存程序的声明（**declaration**），称为头文件。另一个文件用于保存程序的实现（**implementation**），称为定义（**definition**）文件。

C++/C 程序的头文件以“**.h**”为后缀，**C** 程序的定义文件以“**.c**”为后缀，**C++**程序的定义文件通常以“**.cpp**”为后缀（也有一些系统以“**.cc**”或“**.cxx**”为后缀）。

1.1 版权和版本的声明

版权和版本的声明位于头文件和定义文件的开头（参见示例 **1-1**），主要内容有：

（**1**）版权信息。

（**2**）文件名称，标识符，摘要。

（**3**）当前版本号，作者/修改者，完成日期。

（**4**）版本历史信息。

```
/*
* Copyright (c) 2001,上海贝尔有限公司网络应用事业部
* All rights reserved.
*
* 文件名称：filename.h
* 文件标识：见配置管理计划书
* 摘　　要：简要描述本文件的内容
*
* 当前版本：1.1
* 作　　者：输入作者（或修改者）名字
* 完成日期：2001年7月20日
*
* 取代版本：1.0
* 原作者　：输入原作者（或修改者）名字
* 完成日期：2001年5月10日
*/
```

示例1-1 版权和版本的声明

1.2 头文件的结构

头文件由三部分内容组成：

（1）头文件开头处的版权和版本声明（参见示例 1-1）。

（2）预处理块。

（3）函数和类结构声明等。

假设头文件名称为 graphics.h，头文件的结构参见示例 1-2。

- 【规则 1-2-1】为了防止头文件被重复引用，应当用 ifndef/define/endif 结构产生预处理块。
- 【规则 1-2-2】用 #include <filename.h> 格式来引用标准库的头文件（编译器将从标准库目录开始搜索）。
- 【规则 1-2-3】用 #include “filename.h” 格式来引用非标准库的头文件（编译器将从用户的工作目录开始搜索）。

✧ 【建议 1-2-1】头文件中只存放“声明”而不存放“定义”

在 C++ 语法中，类的成员函数可以在声明的同时被定义，并且自动成为内联函数。这虽然会带来书写上的方便，但却造成了风格不一致，弊大于利。建议将成员函数的定义与声明分开，不论该函数体有多么小。

✧ 【建议 1-2-2】不提倡使用全局变量，尽量不要在头文件中出现象 extern int value 这类声明。

```
// 版权和版本声明见示例 1-1，此处省略。

#ifndef   GRAPHICS_H  // 防止 graphics.h 被重复引用
#define   GRAPHICS_H

#include <math.h>       // 引用标准库的头文件
…
#include “myheader.h”    // 引用非标准库的头文件
…
void Function1(…);  // 全局函数声明
…
class Box            // 类结构声明
{
…
};
#endif
```

示例 1-2 C++/C 头文件的结构

1.3 定义文件的结构

定义文件有三部分内容：

（1） 定义文件开头处的版权和版本声明（参见示例 1-1）。

（2） 对一些头文件的引用。

（3） 程序的实现体（包括数据和代码）。

假设定义文件的名称为 graphics.cpp，定义文件的结构参见示例 1-3。

```
// 版权和版本声明见示例 1-1，此处省略。

#include “graphics.h” // 引用头文件
…

// 全局函数的实现体
void Function1(…)
{
  …
}

// 类成员函数的实现体
void Box::Draw(…)
{
  …
}
```

示例 1-3 C++/C 定义文件的结构

1.4 头文件的作用

早期的编程语言如 Basic、Fortran 没有头文件的概念，C++/C 语言的初学者虽然会用使用头文件，但常常不明其理。这里对头文件的作用略作解释：

（1）通过头文件来调用库功能。在很多场合，源代码不便（或不准）向用户公布，只要向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库功能，而不必关心接口怎么实现的。编译器会从库中提取相应的代码。

（2）头文件能加强类型安全检查。如果某个接口被实现或被使用时，其方式与头文件中的声明不一致，编译器就会指出错误，这一简单的规则能大大减轻程序员调试、改错的负担。

1.5 目录结构

如果一个软件的头文件数目比较多（如超过十个），通常应将头文件和定义文件分别保存于不同的目录，以便于维护。

例如可将头文件保存于 **include** 目录，将定义文件保存于 **source** 目录（可以是多级目录）。

如果某些头文件是私有的，它不会被用户的程序直接引用，则没有必要公开其“声明”。为了加强信息隐藏，这些私有的头文件可以和定义文件存放于同一个目录。

第 2 章 程序的版式

版式虽然不会影响程序的功能，但会影响可读性。程序的版式追求清晰、美观，是程序风格的重要构成因素。

可以把程序的版式比喻为“书法”。好的“书法”可让人对程序一目了然，看得兴致勃勃。差的程序“书法”如螃蟹爬行，让人看得索然无味，更令维护者烦恼有加。请程序员们学习程序的“书法”，弥补大学计算机教育的漏洞，实在很有必要。

2.1 空行

空行起着分隔程序段落的作用。空行得体（不过多也不过少）将使程序的布局更加清晰。空行不会浪费内存，虽然打印含有空行的程序是会多消耗一些纸张，但是值得。所以不要舍不得用空行。

- 【规则 2-1-1】在每个类声明之后、每个函数定义结束之后都要加空行。参见示例 2-1（a）
- 【规则 2-1-2】在一个函数体内，逻揖上密切相关的语句之间不加空行，其它地方应加空行分隔。参见示例 2-1（b ）

```
// 空行
void Function1(…)
{
  …
}
// 空行
void Function2(…)
{
  …
}
// 空行
void Function3(…)
{
  …
}
```

示例 2-1(a) 函数之间的空行

```
// 空行
while (condition)
{
  statement1;
  // 空行
  if (condition)
  {
     statement2;
  }
  else
  {
     statement3;
  }
  // 空行
  statement4;
}
```

示例 2-1(b) 函数内部的空行

2.2 代码行

- 【规则 2-2-1】一行代码只做一件事情，如只定义一个变量，或只写一条语句。这样的代码容易阅读，并且方便于写注释。
- 【规则 2-2-2】if、for、while、do 等语句自占一行，执行语句不得紧跟其后。不论执行语句有多少都要加{}。这样可以防止书写失误。

示例 2-2（a）为风格良好的代码行，示例 2-2（b）为风格不良的代码行。

示例 2-2(a) 风格良好的代码行	示例 2-2(b) 风格不良的代码行

`int width;   // 宽度`
`int height;  // 高度`
`int depth;   // 深度`
	`int width, height, depth; // 宽度高度深度`

`x = a + b;`
`y = c + d;`
`z = e + f;`
	`X ＝ a + b;   y = c + d;  z = e + f;`

`if (width < height)`
`{`
`  dosomething();`
`}`
	`if (width < height) dosomething();`

`for (initialization; condition; update)`
`{`
`     dosomething();`
`}`
`// 空行`
`other();`
	`for (initialization; condition; update)`
	`     dosomething();`
	`other();`

示例 2-2(a) 风格良好的代码行　　　　示例 2-2(b) 风格不良的代码行

✧ 【建议 2-2-1】尽可能在定义变量的同时初始化该变量（就近原则）

如果变量的引用处和其定义处相隔比较远，变量的初始化很容易被忘记。如果引用了未被初始化的变量，可能会导致程序错误。本建议可以减少隐患。例如

```
int width = 10;    // 定义并初绐化 width
int height = 10;  // 定义并初绐化 height
int depth = 10;    // 定义并初绐化 depth
```

2.3 代码行内的空格

- 【规则 2-3-1】关键字之后要留空格。象 const、virtual、inline、case 等关键字之后至少要留一个空格，否则无法辨析关键字。象 if、for、while 等关键字之后应留一个空格再跟左括号‘（’，以突出关键字。
- 【规则 2-3-2】函数名之后不要留空格，紧跟左括号‘（’，以与关键字区别。
- 【规则 2-3-3】‘（’向后紧跟，‘）’、‘，’、‘;’向前紧跟，紧跟处不留空格。
- 【规则 2-3-4】‘，’之后要留空格，如 Function(x, y, z)。如果‘;’不是一行的结束符号，其后要留空格，如 for (initialization; condition; update)。
- 【规则 2-3-5】赋值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符，如“=”、“+=” “>=”、“<=”、“+”、“*”、“%”、“&&”、“||”、“<<”,“^”等二元操作符的前后应当加空格。
- 【规则 2-3-6】一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。
- 【规则 2-3-7】象“[]”、“.”、“->”这类操作符前后不加空格。

✧ 【建议 2-3-1】对于表达式比较长的 for 语句和 if 语句，为了紧凑起见可以适当地去掉一些空格，如 for (i=0; i<10; i++)和 if ((a<=b) && (c<=d))

```
void Func1(int x, int y, int z);          // 良好的风格
void Func1 (int x,int y,int z);           // 不良的风格

if (year >= 2000)                         // 良好的风格
if(year>=2000)                            // 不良的风格
if ((a>=b) && (c<=d))                     // 良好的风格
if(a>=b&&c<=d)                            // 不良的风格

for (i=0; i<10; i++)                      // 良好的风格
for(i=0;i<10;i++)                         // 不良的风格
for (i = 0; I < 10; i ++)                 // 过多的空格

x = a < b ? a : b;                        // 良好的风格
x=a<b?a:b;                                // 不好的风格

int *x = &y;                              // 良好的风格
int * x = & y;                            // 不良的风格

array[5] = 0;                             // 不要写成 array [ 5 ] = 0;
a.Function();                             // 不要写成 a . Function();
b->Function();                            // 不要写成 b -> Function();
```

示例 2-3 代码行内的空格

2.4 对齐

- 【规则 2-4-1】程序的分界符‘{’和‘}’应独占一行并且位于同一列，同时与引用它们的语句左对齐。
- 【规则 2-4-2】{ }之内的代码块在‘{’右边数格处左对齐。

示例 2-4（a）为风格良好的对齐，示例 2-4（b）为风格不良的对齐。

<table>
<tr><td><pre>
void Function(int x)
{
  … // program code
}
</pre></td><td><pre>
void Function(int x){
  … // program code
}
</pre></td></tr>
<tr><td><pre>
if (condition)
{
  … // program code
}
else
{
  … // program code
}
</pre></td><td><pre>
if (condition){
  … // program code
}
else {
  … // program code
}
</pre></td></tr>
<tr><td><pre>
for (initialization; condition; update)
{
  … // program code
}
</pre></td><td><pre>
for (initialization; condition; update){
  … // program code
}
</pre></td></tr>
<tr><td><pre>
While (condition)
{
  … // program code
}
</pre></td><td><pre>
while (condition){
  … // program code
}
</pre></td></tr>
<tr><td>如果出现嵌套的 {}，则使用缩进对齐，如：<pre>
    {
       …
         {
           …
         }
      …
    }
</pre></td><td></td></tr>
</table>

示例 2-4(a) 风格良好的对齐　　　　示例 2-4(b) 风格不良的对齐

2.5 长行拆分

- 【规则 2-5-1】代码行最大长度宜控制在 70 至 80 个字符以内。代码行不要过长，否则眼睛看不过来，也不便于打印。
- 【规则 2-5-2】长表达式要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。

```
if ((very_longer_variable1 >= very_longer_variable12)
    && (very_longer_variable3 <= very_longer_variable14)
    && (very_longer_variable5 <= very_longer_variable16))
{
     dosomething();
}
```

```
virtual CMatrix CMultiplyMatrix (CMatrix leftMatrix,
                                 CMatrix rightMatrix);
```

```
for (very_longer_initialization;
     very_longer_condition;
     very_longer_update)
{
    dosomething();
}
```

示例 2-5 长行的拆分

2.6 修饰符的位置

修饰符 * 和 & 应该靠近数据类型还是该靠近变量名，是个有争议的活题。

若将修饰符 * 靠近数据类型，例如：int* x; 从语义上讲此写法比较直观，即 x 是 int 类型的指针。

上述写法的弊端是容易引起误解，例如：int* x, y; 此处 y 容易被误解为指针变量。虽然将 x 和 y 分行定义可以避免误解，但并不是人人都愿意这样做。

- 【规则 2-6-1】应当将修饰符 * 和 & 紧靠变量名

例如：

```
char  *name;
int   *x, y;   // 此处 y 不会被误解为指针
```

2.7 注释

C 语言的注释符为“/*…*/”。C++语言中，程序块的注释常采用“/*…*/”，行注释一般采用“//…”。注释通常用于：

（1）版本、版权声明；

（2）函数接口说明；

（3）重要的代码行或段落提示。

虽然注释有助于理解代码，但注意不可过多地使用注释。参见示例 2-6。

- 【规则 2-7-1】注释是对代码的“提示”，而不是文档。程序中的注释不可喧宾夺主，注释太多了会让人眼花缭乱。注释的花样要少。
- 【规则 2-7-2】如果代码本来就是清楚的，则不必加注释。否则多此一举，令人厌烦。例如

```
i++;    // i 加 1，多余的注释
```

- 【规则 2-7-3】边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。
- 【规则 2-7-4】注释应当准确、易懂，防止注释有二义性。错误的注释不但无益反而有害。
- 【规则 2-7-5】尽量避免在注释中使用缩写，特别是不常用缩写。
- 【规则 2-7-6】注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。
- 【规则 2-7-8】当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读。

<table>
<tr>
<td>

```
/*
* 函数介绍：
* 输入参数：
* 输出参数：
* 返回值  ：
*/
void Function(float x, float y, float z)
{
   …
}
```

</td>
<td>

```
if (…)
{
   …
   while (…)
   {
      …
   } // end of while
   …
} // end of if
```

</td>
</tr>
</table>

示例 2-6 程序的注释

2.8 类的版式

类可以将数据和函数封装在一起，其中函数表示了类的行为（或称服务）。类提供关键字 **public**、**protected** 和 **private**，分别用于声明哪些数据和函数是公有的、受保护的或者是私有的。这样可以达到信息隐藏的目的，即让类仅仅公开必须要让外界知道的内容，而隐藏其它一切内容。我们不可以滥用类的封装功能，不要把它当成火锅，什么东西都往里扔。

类的版式主要有两种方式：

（1）将 **private** 类型的数据写在前面，而将 **public** 类型的函数写在后面，如示例 **8-3**（**a**）。采用这种版式的程序员主张类的设计"以数据为中心"，重点关注类的内部结构。

（2）将 **public** 类型的函数写在前面，而将 **private** 类型的数据写在后面，如示例 **8.3**（**b**）采用这种版式的程序员主张类的设计"以行为为中心"，重点关注的是类应该提供什么样的接口（或服务）。

很多 **C++**教课书受到 **Biarne Stroustrup** 第一本著作的影响，不知不觉地采用了"以数据为中心"的书写方式，并不见得有多少道理。

我建议读者采用"以行为为中心"的书写方式，即首先考虑类应该提供什么样的函数。这是很多人的经验——"这样做不仅让自己在设计类时思路清晰，而且方便别人阅读。因为用户最关心的是接口，谁愿意先看到一堆私有数据成员！"

```
class A
{
  private:
    int    i, j;
    float  x, y;
    …
  public:
    void Func1(void);
    void Func2(void);
    …
}
```

示例 **8.3(a)** 以数据为中心版式

```
class A
{
  public:
    void Func1(void);
    void Func2(void);
    …
  private:
    int    i, j;
    float  x, y;
    …
}
```

示例 **8.3(b)** 以行为为中心的版式

第 3 章 命名规则

比较著名的命名规则当推 **Microsoft** 公司的“匈牙利”法，该命名规则的主要思想是“在变量和函数名中加入前缀以增进人们对程序的理解”。例如所有的字符变量均以 ch 为前缀，若是指针变量则追加前缀 p。如果一个变量由 ppch 开头，则表明它是指向字符指针的指针。

“匈牙利”法最大的缺点是烦琐，例如

```
int    i,  j,  k;
float  x,  y,  z;
```

倘若采用“匈牙利”命名规则，则应当写成

```
int    iI,  iJ,  ik;  // 前缀 i 表示 int 类型
float  fX,  fY,  fZ;  // 前缀 f 表示 float 类型
```

如此烦琐的程序会让绝大多数程序员无法忍受。

据考察，没有一种命名规则可以让所有的程序员赞同，程序设计教科书一般都不指定命名规则。命名规则对软件产品而言并不是“成败悠关”的事，我们不要化太多精力试图发明世界上最好的命名规则，而应当制定一种令大多数项目成员满意的命名规则，并在项目中贯彻实施。

3.1 共性规则

本节论述的共性规则是被大多数程序员采纳的，我们应当在遵循这些共性规则的前提下，再扩充特定的规则，如 **3.2** 节。

- 【规则 **3-1-1**】标识符应当直观且可以拼读，可望文知意，不必进行“解码”。

标识符最好采用英文单词或其组合，便于记忆和阅读。切忌使用汉语拼音来命名。程序中的英文单词一般不会太复杂，用词应当准确。例如不要把 **CurrentValue** 写成 **NowValue**。

- 【规则 **3-1-2**】标识符的长度应当符合“min-length && max-information”原则。

几十年前老 **ANSI C** 规定名字不准超过 **6** 个字符，现今的 C++/C 不再有此限制。一般来说，长名字能更好地表达含义，所以函数名、变量名、类名长达十几个字符不足为怪。那么名字是否越长约好？不见得！例如变量名 maxval 就比 maxValueUntilOverflow 好用。单字符的名字也是有用的，常见的如 i,j,k,m,n,x,y,z 等，它们通常可用作函数内的局部变量。

- 【规则 **3-1-3**】命名规则尽量与所采用的操作系统或开发工具的风格保持一致。

例如 **Windows** 应用程序的标识符通常采用“大小写”混排的方式，如 **AddChild**。而 **Unix** 应用程序的标识符通常采用“小写加下划线”的方式，如 **add_child**。别把这两

类风格混在一起用。

- 【规则 3-1-4】程序中不要出现仅靠大小写区分的相似的标识符。
  例如：

```
int   x,  X;        // 变量 x 与 X 容易混淆
void foo(int x);   // 函数 foo 与 FOO 容易混淆
void FOO(float x);
```

- 【规则 3-1-5】程序中不要出现标识符完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但会使人误解。

- 【规则 3-1-6】变量的名字应当使用“名词”或者“形容词＋名词”。
  例如：

```
float   value;
float   oldValue;
float   newValue;
```

- 【规则 3-1-7】全局函数的名字应当使用“动词”或者“动词＋名词”（动宾词组）。类的成员函数应当只使用“动词”，被省略掉的名词就是对象本身。
  例如：

```
DrawBox();          // 全局函数
box->Draw();            // 类的成员函数
```

- 【规则 3-1-8】用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。
  例如：

```
int  minValue;
int  maxValue;

int  SetValue(…);
int  GetValue(…);
```

✧ 【建议 3-1-1】尽量避免名字中出现数字编号，如 Value1,Value2 等，除非逻辑上的确需要编号。这是为了防止程序员偷懒，不肯为命名动脑筋而导致产生无意义的名字（因为用数字编号最省事）。

3.2 简单的 Windows 应用程序命名规则

作者对“匈牙利”命名规则做了合理的简化，下述的命名规则简单易用，比较适合于 Windows 应用软件的开发。

- **【规则 3-2-1】类名和函数名用大写字母开头的单词组合而成。**

例如：

```
class Node;                // 类名
class LeafNode;                // 类名
void  Draw(void);          // 函数名
void  SetValue(int value);// 函数名
```

- **【规则 3-2-2】变量和参数用小写字母开头的单词组合而成。**

例如：

```
BOOL flag;
int  drawMode;
```

- **【规则 3-2-3】常量全用大写的字母，用下划线分割单词。**

例如：

```
const int MAX = 100;
const int MAX_LENGTH = 100;
```

- **【规则 3-2-4】静态变量加前缀 s_（表示 static）。**

例如：

```
void Init(…)
{
    static int s_initValue; //  静态变量
    …
}
```

- **【规则 3-2-5】如果不得已需要全局变量，则使全局变量加前缀 g_（表示 global）。**

例如：

```
int g_howManyPeople;    //  全局变量
int g_howMuchMoney;     //  全局变量
```

- **【规则 3-2-6】类的数据成员加前缀 m_（表示 member），这样可以避免数据成员与成员函数的参数同名。**

例如：

```
void Object::SetValue(int width, int height)
{
    m_width = width;
    m_height = height;
}
```

- **【规则 3-2-7】为了防止某一软件库中的一些标识符和其它软件库中的冲突，可以为各种标识符加上能反映软件性质的前缀。例如三维图形标准 OpenGL 的所有库函数均以 gl 开头，所有常量（或宏定义）均以 GL 开头。**

3.3 简单的 Unix 应用程序命名规则

第 4 章 表达式和基本语句

读者可能怀疑：连 if、for、while、goto、switch 这样简单的东西也要探讨编程风格，是不是小题大做？

我真的发觉很多程序员用隐含错误的方式写表达式和基本语句，我自己也犯过类似的错误。

表达式和语句都属于 C++/C 的短语结构语法。它们看似简单，但使用时隐患比较多。本章归纳了正确使用表达式和语句的一些规则与建议。

4.1 运算符的优先级

C++/C 语言的运算符有数十个，运算符的优先级与结合律如表 **4-1** 所示。注意一元运算符 + - * 的优先级高于对应的二元运算符。

优先级	运算符	结合律
从高到低排列	( ) [ ] -> .	从左至右
	! ~ ++ -- （类型） sizeof + - * &	从右至左
	* / %	从左至右
	+ -	从左至右
	<< >>	从左至右
	< <= > >=	从左至右
	== !=	从左至右
	&	从左至右
	^	从左至右
	|	从左至右
	&&	从左至右
	||	从右至左
	?:	从右至左
	= += -= *= /= %= &= ^= |= <<= >>=	从左至右

表 **4-1** 运算符的优先级与结合律

- 【规则 **4-1-1**】如果代码行中的运算符比较多，用括号确定表达式的操作顺序，避免使用默认的优先级。

由于将表 **4-1** 熟记是比较困难的，为了防止产生歧义并提高可读性，应当用括号确定表达式的操作顺序。例如：

word = (high << 8) | low

if ((a | b) && (a & c))

4.2 复合表达式

如 **a = b = c = 0** 这样的表达式称为复合表达式。允许复合表达式存在的理由是：(1) 书写简洁；(2) 可以提高编译效率。但要防止滥用复合表达式。

- 【规则 4-2-1】不要编写太复杂的复合表达式。

  例如：

  ```
  i = a >= b && c < d && c + f <= g + h ;  // 复合表达式过于复杂
  ```

- 【规则 4-2-2】不要有多用途的复合表达式。

  例如：

  ```
  d = (a = b + c) + r ;
  ```

  该表达式既求 **a** 值又求 **d** 值。应该拆分为两个独立的语句：

  ```
  a = b + c;
  d = a + r;
  ```

- 【规则 4-2-3】不要把程序中的复合表达式与“真正的数学表达式”混淆。

  例如：

  ```
  if (a < b < c)         // a < b < c 是数学表达式而不是程序表达式
  ```

  并不表示

  ```
  if ((a<b) && (b<c))
  ```

  而是成了令人费解的

  ```
  if ( (a<b)<c )
  ```

4.3 if 语句

if 语句是 C++/C 语言中最简单、最常用的语句，然而很多程序员用隐含错误的方式写 if 语句。本节以“与零值比较”为例，展开讨论。

4.3.1 布尔变量与零值比较

- 【规则 4-3-1】不可将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较。

  根据布尔类型的语义，零值为“假”(记为 FALSE)，任何非零值都是“真”(记为 TRUE)。TRUE 的值究竟是什么并没有统一的标准。例如 Visual C++ 将 TRUE 定义为 1，而 Visual Basic 则将 TRUE 定义为-1。

  假设布尔变量名字为 flag，它与零值比较的标准 if 语句如下：

  ```
  if (flag)  // 表示 flag 为真
  ```

```
if (!flag) // 表示 flag 为假
```

其它的用法都属于不良风格，例如：

```
if (flag == TRUE)
if (flag == 1 )
if (flag == FALSE)
if (flag == 0)
```

4.3.2 整型变量与零值比较

- 【**规则 4-3-2**】应当将整型变量用“==”或“!=”直接与 0 比较。

假设整型变量的名字为 value，它与零值比较的标准 **if** 语句如下：

```
if (value == 0)
if (value != 0)
```

不可模仿布尔变量的风格而写成

```
if (value)     // 会让人误解 value 是布尔变量
if (!value)
```

4.3.3 浮点变量与零值比较

- 【**规则 4-3-3**】不可将浮点变量用“==”或“!=”与任何数字比较。

千万要留意，无论是 float 还是 double 类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“!=”与数字比较，应该设法转化成“>=”或“<=”形式。

假设浮点变量的名字为 x，应当将

```
if (x == 0.0)  // 隐含错误的比较
```

转化为

```
if ((x>=-EPSINON) && (x<=EPSINON))
```

其中 EPSINON 是允许的误差（即精度）。

4.3.4 指针变量与零值比较

- 【**规则 4-3-4**】应当将指针变量用“==”或“!=”与 NULL 比较。

指针变量的零值是“空”(记为 NULL)。尽管 NULL 的值与 0 相同，但是两者意义不同。假设指针变量的名字为 p，它与零值比较的标准 **if** 语句如下：

```
if (p == NULL) // p 与 NULL 显式比较，强调 p 是指针变量
if (p != NULL)
```

不要写成

```
if (p == 0)    // 容易让人误解 p 是整型变量
if (p != 0)
```

或者

```
if (p)         // 容易让人误解 p 是布尔变量
if (!p)
```

4.3.5 对 if 语句的补充说明

有时候我们可能会看到 if (NULL == p) 这样古怪的格式。不是程序写错了，是程序员为了防止将 if (p == NULL) 误写成 if (p = NULL)，而有意把 p 和 NULL 颠倒。编译器认为 if (p = NULL) 是合法的，但是会指出 if (NULL = p)是错误的，因为 NULL 不能被赋值。

程序中有时会遇到 if/else/return 的组合，应该将如下不良风格的程序

```
if (condition)
    return x;
return y;
```

改写为

```
if (condition)
{
    return x;
}
else
{
    return y;
}
```

或者改写成更加简练的

```
return (condition ? x : y);
```

4.4 循环语句的效率

C++/C 循环语句中，for 语句使用频率最高，while 语句其次，do 语句很少用。本节重点论述循环体的效率。提高循环体效率的基本办法是降低循环体的复杂性。

- 【建议 4-4-1】在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少 CPU 跨切循环层的次数。例如示例 4-4(b)的效率比示例 4-4(a)的高。

```
for (row=0; row<100; row++)
{
    for ( col=0; col<5; col++ )
    {
        sum = sum + a[row][col];
    }
}
```

示例 4-4(a) 低效率：长循环在最外层

```
for (col=0; col<5; col++ )
{
    for (row=0; row<100; row++)
    {
        sum = sum + a[row][col];
    }
}
```

示例 4-4(b) 高效率：长循环在最内层

- 【建议 4-4-2】如果循环体内存在逻辑判断，并且循环次数很大，宜将逻辑判断移到

循环体的外面。示例 4-4(c)的程序比示例 4-4(d)多执行了 N-1 次逻辑判断。并且由于前者老要进行逻辑判断，打断了循环“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。如果 N 非常大，最好采用示例 4-4(d)的写法，可以提高效率。如果 N 非常小，两者效率差别并不明显，采用示例 4-4(c)的写法比较好，因为程序更加简洁。

```
for (i=0; i<N; i++)
{
  if (condition)
      DoSomething();
  else
      DoOtherthing();
}
```

表 4-4(c) 效率低但程序简洁

```
if (condition)
{
    for (i=0; i<N; i++)
      DoSomething();
}
else
{
    for (i=0; i<N; i++)
      DoOtherthing();
}
```

表 4-4(d) 效率高但程序不简洁

4.5 for 语句的循环控制变量

- 【规则 4-5-1】不可在 **for** 循环体内修改循环变量，防止 **for** 循环失去控制。

- 【建议 4-5-1】建议 for 语句的循环控制变量的取值采用“半开半闭区间”写法。

示例 4-5(a)中的 x 值属于半开半闭区间“0 =< x < N”，起点到终点的间隔为 N，循环次数为 N。

示例 4-5(b)中的 x 值属于闭区间“0 =< x <= N-1”，起点到终点的间隔为 N-1，循环次数为 N。

相比之下，示例 4-5(a)的写法更加直观，尽管两者的功能是相同的。

```
for (int x=0; x<N; x++)
{
  …
}
```

示例 4-5(a) 循环变量属于半开半闭区间

```
for (int x=0; x<=N-1; x++)
{
  …
}
```

示例 4-5(b) 循环变量属于闭区间

4.6 switch 语句

有了 if 语句为什么还要 switch 语句？

switch 是多分支选择语句，而 if 语句只有两个分支可供选择。虽然可以用嵌套的 if 语句来实现多分支选择，但那样的程序冗长难读。这是 switch 语句存在的理由。

switch 语句的基本格式是：

```
switch (variable)
{
    case value1 :  …
                   break;
    case value2 :  …
                   break;
    …
      default :  …
                   break;
}
```

- 【规则 **4-6-1**】每个 case 语句的结尾不要忘了加 break，否则将导致多个分支重叠（除非有意使多个分支重叠）。
- 【规则 **4-6-2**】不要忘记最后那个 default 分支。即使程序真的不需要 default 处理，也应该保留语句 default : break; 这样做并非多此一举，而是为了防止别人误以为你忘了 default 处理。

4.7 goto 语句

自从提倡结构化设计以来，goto 就成了有争议的语句。首先，由于 goto 语句可以灵活跳转，如果不加限制，它的确会破坏结构化设计风格。其次，goto 语句经常带来错误或隐患。它可能跳过了某些对象的构造、变量的初始化、重要的计算等语句，例如：

```
goto state;
String s1, s2; // 被 goto 跳过
int sum = 0;   // 被 goto 跳过
…
state:
…
```

如果编译器不能发觉此类错误，每用一次 goto 语句都可能留下隐患。

很多人建议废除 C++/C 的 goto 语句，以绝后患。但实事求是地说，错误是程序员自己造成的，不是 goto 的过错。goto 语句至少有一处可显神通，它能从多重循环体中咻地一下子跳到外面，用不着写很多次的 break 语句; 例如

```
{ …
    { …
        { …
            goto error;
```

```
            }
        }
    }
    error:
    …
```

就象楼房着火了，来不及从楼梯一级一级往下走，可从窗口跳出火坑。所以我们主张少用、慎用 goto 语句，而不是禁用。

第 5 章 常量

常量是一种标识符，它的值在运行期间恒定不变。C 语言用 #define 来定义常量（称为宏常量）。C++ 语言除了 #define 外还可以用 const 来定义常量（称为 const 常量）。

5.1 为什么需要常量

如果不使用常量，直接在程序中填写数字或字符串，将会有什么麻烦？

(1) 程序的可读性（可理解性）变差。程序员自己会忘记那些数字或字符串是什么意思，用户则更加不知它们从何处来、表示什么。

(2) 在程序的很多地方输入同样的数字或字符串，难保不发生书写错误。

(3) 如果要修改数字或字符串，则会在很多地方改动，既麻烦又容易出错。

- 【规则 5-1-1】 尽量使用含义直观的常量来表示那些将在程序中多次出现的数字或字符串。

  例如：

```
#define         MAX   100       /*  C 语言的宏常量  */
const int       MAX = 100;      //  C++ 语言的 const 常量
const float     PI = 3.14159;   //  C++ 语言的 const 常量
```

5.2 const 与 #define 的比较

C++ 语言可以用 const 来定义常量，也可以用 #define 来定义常量。但是前者比后者有更多的优点：

(1) const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误（边际效应）。

(2) 有些集成化的调试工具可以对 const 常量进行调试，但是不能对宏常量进行调试。

- 【规则 5-2-1】在 C++ 程序中只使用 const 常量而不使用宏常量，即 const 常量完全取代宏常量。

5.3 常量定义规则

- 【规则 5-3-1】需要对外公开的常量放在头文件中，不需要对外公开的常量放在定义文件的头部。为便于管理，可以把不同模块的常量集中存放在一个公共的头文件中。

- **【规则 5-3-2】如果某一常量与其它常量密切相关，应在定义中包含这种关系，而不应给出一些孤立的值。**

  例如：

```
const   float    RADIUS = 100;
const   float    DIAMETER = RADIUS * 2;
```

5.4 类中的常量

有时我们希望某些常量只在类中有效。由于#define 定义的宏常量是全局的，不能达到目的，于是想当然地觉得应该用 const 修饰数据成员来实现。const 数据成员的确是存在的，但其含义却不是我们所期望的。const 数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的，因为类可以创建多个对象，不同的对象其 const 数据成员的值可以不同。

不能在类声明中初始化 const 数据成员。以下用法是错误的，因为类的对象未被创建时，编译器不知道 SIZE 的值是什么。

```
class A
{…
    const int SIZE = 100;  // 错误，企图在类声明中初始化 const 数据成员
    int array[SIZE];       // 错误，未知的 SIZE
};
```

const 数据成员的初始化只能在类构造函数的初始化表中进行，例如

```
class A
{…
    A(int size);       // 构造函数
    const int SIZE ;
};
A::A(int size) : SIZE(size)   // 构造函数的初始化表
{
  …
}
A  a(100); // 对象 a 的 SIZE 值为 100
A  b(200); // 对象 b 的 SIZE 值为 200
```

怎样才能建立在整个类中都恒定的常量呢？别指望 const 数据成员了，应该用类中的枚举常量来实现。例如

```
class A
{…
    enum { SIZE1 = 100, SIZE2 = 200}; // 枚举常量
```

```
        int array1[SIZE1];
        int array2[SIZE2];
    };
```

枚举常量不会占用对象的存储空间，它们在编译时被全部求值。枚举常量的缺点是：它的隐含数据类型是整数，其最大值有限，且不能表示浮点数（如 PI=3.14159）。

第 6 章 函数设计

函数是 C++/C 程序的基本功能单元，其重要性不言而喻。函数设计的细微缺点很容易导致该函数被错用，所以光使函数的功能正确是不够的。本章重点论述函数的接口设计和内部实现的一些规则。

函数接口的两个要素是参数和返回值。C 语言中，函数的参数和返回值的传递方式有两种：值传递（pass by value）和指针传递（pass by pointer）。C++ 语言中多了引用传递（pass by reference）。由于引用传递的性质象指针传递，而使用方式却象值传递，初学者常常迷惑不解，容易引起混乱，请先阅读 6.6 节“引用与指针的比较”。

6.1 参数的规则

- 【规则 6-1-1】参数的书写要完整，不要贪图省事只写参数的类型而省略参数名字。如果函数没有参数，则用 void 填充。

  例如：

```
void SetValue(int width, int height);    // 良好的风格
void SetValue(int, int);                 // 不良的风格
float GetValue(void);     // 良好的风格
float GetValue();         // 不良的风格
```

- 【规则 6-1-2】参数命名要恰当，顺序要合理。

例如编写字符串拷贝函数 StringCopy，它有两个参数。如果把参数名字起为 str1 和 str2，例如

```
void StringCopy(char *str1, char *str2);
```

那么我们很难搞清楚究竟是把 str1 拷贝到 str2 中，还是刚好倒过来。

可以把参数名字起得更有意义，如叫 strSource 和 strDestination。这样从名字上就可以看出应该把 strSource 拷贝到 strDestination。

还有一个问题，这两个参数那一个该在前那一个该在后？参数的顺序要遵循程序员的习惯。一般地，应将目的参数放在前面，源参数放在后面。

如果将函数声明为：

```
void StringCopy(char *strSource, char *strDestination);
```

别人在使用时可能会不假思索地写成如下形式：

```
char str[20];
StringCopy(str, “Hello World”);// 参数顺序颠倒
```

- 【规则 6-1-3】如果参数是指针，且仅作输入用，则应在类型前加 const，以防止该指针在函数体内被意外修改。

  例如：

```
void StringCopy(char *strDestination，const char *strSource);
```

- 【规则 6-1-4】如果输入参数以值传递的方式传递对象，则宜改用“ const & ”方式来传递，这样可以省去临时对象的构造和析构过程，从而提高效率。

✧ 【建议 6-1-1】避免函数有太多的参数，参数个数尽量控制在 5 个以内。如果参数太多，在使用时容易将参数类型或顺序搞错。

✧ 【建议 6-1-2】尽量不要使用类型和数目不确定的参数。

C 标准库函数 printf 是采用不确定参数的典型代表，其原型为：

```
int printf(const chat *format[, argument]…);
```

这种风格的函数在编译时丧失了严格的类型安全检查。

6.2 返回值的规则

- 【规则 6-2-1】不要省略返回值的类型。

C 语言中，凡不加类型说明的函数，一律自动按整型处理。这样做不会有什么好处，却容易被误解为 void 类型。

C++语言有很严格的类型安全检查，不允许上述情况发生。由于 C++程序可以调用 C 函数，为了避免混乱，规定任何 C++/ C 函数都必须有类型。如果函数没有返回值，那么应声明为 void 类型。

- 【规则 6-2-2】函数名字与返回值类型在语义上不可冲突。

违反这条规则的典型代表是 C 标准库函数 getchar。

例如：

```
char c;
c = getchar();
if (c == EOF)
…
```

按照 getchar 名字的意思，将变量 c 声明为 char 类型是很自然的事情。但不幸的是 getchar 的确不是 char 类型，而是 int 类型，其原型如下：

```
int getchar(void);
```

由于 c 是 char 类型，取值范围是[-128，127]，如果宏 EOF 的值在 char 的取值范围之外，那么 if 语句将总是失败，这种“危险”人们一般哪里料得到！导致本例错误的责任并不在用户，是函数 getchar 误导了使用者。

- 【规则 6-2-3】不要将正常值和错误标志混在一起返回。正常值用输出参数获得，而错误标志用 return 语句返回。

回顾上例，C 标准库函数的设计者为什么要将 getchar 声明为令人迷糊的 int 类型

呢？他会那么傻吗？

在正常情况下，getchar 的确返回单个字符。但如果 getchar 碰到文件结束标志或发生读错误，它必须返回一个标志 EOF。为了区别于正常的字符，只好将 EOF 定义为负数（通常为负 1）。因此函数 getchar 就成了 int 类型。

我们在实际工作中，经常会碰到上述令人为难的问题。为了避免出现误解，我们应该将正常值和错误标志分开。即：正常值用输出参数获得，而错误标志用 return 语句返回。

函数 getchar 可以改写成 `BOOL GetChar(char *c);`

虽然 gechar 比 GetChar 灵活，例如 `putchar(getchar());` 但是如果 getchar 用错了，它的灵活性又有什么用呢？

✧ 【建议 6-2-1】有时候函数原本不需要返回值，但为了增加灵活性如支持链式表达，可以附加返回值。

例如字符串拷贝函数 strcpy 的原型：

```
char *strcpy(char *strDest，const char *strSrc);
```

strcpy 函数将 strSrc 拷贝至输出参数 strDest 中，同时函数的返回值又是 strDest。这样做并非多此一举，可以获得如下灵活性：

```
char str[20];
int  length = strlen( strcpy(str, “Hello World”) );
```

✧ 【建议 6-2-2】如果函数的返回值是一个对象，有些场合用“引用传递”替换“值传递”可以提高效率。而有些场合只能用“值传递”而不能用“引用传递”，否则会出错。

例如：

```
class String
{…
    // 赋值函数
    String & operate=(const String &other);
    // 相加函数，如果没有 friend 修饰则只许有一个右侧参数
    friend String   operate+( const String &s1, const String &s2);
private:
    char *m_data;
}
```

String 的赋值函数 operate = 的实现如下：

```
String & String::operate=(const String &other)
{
    if (this == &other)
        return *this;
    delete m_data;
```

```
        m_data = new char[strlen(other.data)+1];
        strcpy(m_data, other.data);
        return *this;  // 返回的是 *this 的引用，无需拷贝过程
    }
```

对于赋值函数，应当用“引用传递”的方式返回 **String** 对象。如果用“值传递”的方式，虽然功能仍然正确，但由于 **return** 语句要把 *this 拷贝到保存返回值的外部存储单元之中，增加了不必要的开销，降低了赋值函数的效率。例如：

```
String a,b,c;
…
a = b;      // 如果用“值传递”，将产生一次 *this 拷贝
a = b = c;  // 如果用“值传递”，将产生两次 *this 拷贝
```

String 的相加函数 **operate +** 的实现如下：

```
String  operate+(const String &s1, const String &s2)
{
    String temp;
    delete temp.data;  // temp.data 是仅含‘\0’的字符串
    temp.data = new char[strlen(s1.data) + strlen(s2.data) +1];
    strcpy(temp.data, s1.data);
    strcat(temp.data, s2.data);
    return temp;
}
```

对于相加函数，应当用“值传递”的方式返回 **String** 对象。如果改用“引用传递”，那么函数返回值是一个指向局部对象 **temp** 的“引用”。由于 **temp** 在函数结束时被自动销毁，将导致返回的“引用”无效。例如：

```
c = a + b;
```

此时 a + b 并不返回期望值，c 什么也得不到，流下了隐患。

6.3 函数内部实现的规则

不同功能的函数其内部实现各不相同，看起来似乎无法就“内部实现”达成一致的观点。但根据经验，我们可以在函数体的“入口处”和“出口处”从严把关，从而提高函数的质量。

- 【规则 **6-3-1**】在函数体的“入口处”，对参数的有效性进行检查。

  很多程序错误是由非法参数引起的，我们应该充分理解并正确使用“断言”(**assert**)来防止此类错误。详见 **6.5** 节“使用断言”。

- 【规则 6-3-2】在函数体的“出口处”，对 return 语句的正确性和效率进行检查。

如果函数有返回值，那么函数的“出口处”是 return 语句。我们不要轻视 return 语句。如果 return 语句写得不好，函数要么出错，要么效率低下。

注意事项如下：

（1）return 语句不可返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。例如

```
char * Func(void)
{
    char str[] = “hello world”;   // str 的内存位于栈上
    …
    return str;    // 将导致错误
}
```

（2）要搞清楚返回的究竟是“值”、“指针”还是“引用”。

（3）如果函数返回值是一个对象，要考虑 return 语句的效率。例如

```
return String(s1 + s2);
```

这是临时对象的语法，表示“创建一个临时对象并返回它”。不要以为它与“先创建一个局部对象 temp 并返回它的结果”是等价的，如

```
String temp(s1 + s2);
return temp;
```

实质不然，上述代码将发生三件事。首先，temp 对象被创建，同时完成初始化；然后拷贝构造函数把 temp 拷贝到保存返回值的外部存储单元中；最后，temp 在函数结束时被销毁（调用析构函数）。然而“创建一个临时对象并返回它”的过程是不同的，编译器直接把临时对象创建并初始化在外部存储单元中，省去了拷贝和析构的化费，提高了效率。

类似地，我们不要将

```
return int(x + y); // 创建一个临时变量并返回它
```

写成

```
int temp = x + y;
return temp;
```

由于内部数据类型如 int,float,double 的变量不存在构造函数与析构函数，虽然该“临时变量的语法”不会提高多少效率，但是程序更加简洁易读。

6.4 其它建议

- ✧ 【建议 6-4-1】函数的功能要单一，不要设计多用途的函数。
- ✧ 【建议 6-4-2】函数体的规模要小，尽量控制在 50 行代码之内。
- ✧ 【建议 6-4-3】尽量避免函数带有“记忆”功能。相同的输入应当产生相同的输出。

带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某

种“记忆状态”。这样的函数既不易理解又不利于测试和维护。在 C/C++语言中，函数的 static 局部变量是函数的“记忆”存储器。建议尽量少用 static 局部变量，除非必需。

✧ 【建议 6-4-4】不仅要检查输入参数的有效性，还要检查通过其它途径进入函数体内的变量的有效性，例如全局变量、文件句柄等。

✧ 【建议 6-4-5】用于出错处理的返回值一定要清楚，让使用者不容易忽视或误解错误情况。

6.5 使用断言

程序一般分为 Debug 版本和 Release 版本，Debug 版本用于内部调试，Release 版本发行给用户使用。

断言 assert 是仅在 Debug 版本起作用的宏，它用于检查“不应该”发生的情况。示例 6-5 是一个内存复制函数。在运行过程中，如果 assert 的参数为假，那么程序就会中止（一般地还会出现提示对话，说明在什么地方引发了 assert）。

```
void  *memcpy(void *pvTo, const void *pvFrom, size_t size)
{
    assert((pvTo != NULL) && (pvFrom != NULL));  // 使用断言
    byte *pbTo = (byte *) pvTo;       // 防止改变 pvTo 的地址
    byte *pbFrom = (byte *) pvFrom;   // 防止改变 pvFrom 的地址
    while(size -- > 0 )
        *pbTo ++ = *pbFrom ++ ;
    return pvTo;
}
```

示例 6-5 复制不重叠的内存块

assert 不是一个仓促拼凑起来的宏。为了不在程序的 Debug 版本和 Release 版本引起差别，assert 不应该产生任何副作用。所以 assert 不是函数，而是宏。程序员可以把 assert 看成一个在任何系统状态下都可以安全使用的无害测试手段。如果程序在 assert 处终止了，并不是说含有该 assert 的函数有错误，而是调用者出了差错，assert 可以帮助我们找到发生错误的原因。

很少有比跟踪到程序的断言，却不知道该断言的作用更让人沮丧的事了。你化了很多时间，不是为了排除错误，而只是为了弄清楚这个错误到底是什么。有的时候，程序员偶尔还会设计出有错误的断言。所以如果搞不清楚断言检查的是什么，就很难判断错误是出现在程序中，还是出现在断言中。幸运的是这个问题很好解决，只要加上清晰的注释即可。这本是显而易见的事情，可是很少有程序员这样做。这好比一个人在森林里，看到树上钉着一块“危险”的大牌子。但危险到底是什么？树要倒？有废井？有野兽？除非告诉人们“危险”是什么，否则这个警告牌难以起到积极有效的作用。难以理解的断言常常被程序员忽略，甚至被删除。[Maguire, p8-p30]

- 【规则 6-5-1】使用断言捕捉不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，后者是必然存在的并且是一定要作出处理的。
- 【规则 6-5-2】在函数的入口处，使用断言检查参数的有效性（合法性）。
- 【建议 6-5-1】在编写函数时，要进行反复的考查，并且自问："我打算做哪些假定？"一旦确定了的假定，就要使用断言对假定进行检查。
- 【建议 6-5-2】一般教科书都鼓励程序员们进行防错设计，但要记住这种编程风格可能会隐瞒错误。当进行防错设计时，如果"不可能发生"的事情的确发生了，则要使用断言进行报警。

6.6 引用与指针的比较

引用是 C++中的概念，初学者容易把引用和指针混淆一起。一下程序中，n 是 m 的一个引用（reference），m 是被引用物（referent）。

```
int m;
int &n = m;
```

n 相当于 m 的别名（绰号），对 n 的任何操作就是对 m 的操作。例如有人名叫王小毛，他的绰号是"三毛"。说"三毛"怎么怎么的，其实就是对王小毛说三道四。所以 n 既不是 m 的拷贝，也不是指向 m 的指针，其实 n 就是 m 它自己。

引用的一些规则如下：

（1）引用被创建的同时必须被初始化（指针则可以在任何时候被初始化）。
（2）不能有 NULL 引用，引用必须与合法的存储单元关联（指针则可以是 NULL）。
（3）一旦引用被初始化，就不能改变引用的关系（指针则可以随时改变所指的对象）。

以下示例程序中，k 被初始化为 i 的引用。语句 k = j 并不能将 k 修改成为 j 的引用，只是把 k 的值改变成为 6。由于 k 是 i 的引用，所以 i 的值也变成了 6。

```
int i = 5;
int j = 6;
int &k = i;
k = j; // k 和 i 的值都变成了 6;
```

上面的程序看起来象在玩文字游戏，没有体现出引用的价值。引用的主要功能是传递函数的参数和返回值。C++语言中，函数的参数和返回值的传递方式有三种：值传递、指针传递和引用传递。

以下是"值传递"的示例程序。由于 Func1 函数体内的 x 是外部变量 n 的一份拷贝，改变 x 的值不会影响 n，所以 n 的值仍然是 0。

```
void Func1(int x)
{
    x = x + 10;
}
…
```

```
int n = 0;
Func1(n);
cout << “n = ” << n << endl;    // n = 0
```

以下是“指针传递”的示例程序。由于 Func2 函数体内的 x 是指向外部变量 n 的指针，改变该指针的内容将导致 n 的值改变，所以 n 的值成为 10。

```
void Func2(int *x)
{
    (* x) = (* x) + 10;
}
…
int n = 0;
Func2(&n);
cout << “n = ” << n << endl;        // n = 10
```

以下是“引用传递”的示例程序。由于 Func3 函数体内的 x 是外部变量 n 的引用，x 和 n 是同一个东西，改变 x 等于改变 n，所以 n 的值成为 10。

```
void Func3(int &x)
{
    x = x + 10;
}
…
int n = 0;
Func3(n);
cout << “n = ” << n << endl;        // n = 10
```

对比上述三个示例程序，会发现“引用传递”的性质象“指针传递”，而书写方式象“值传递”。实际上“引用”可以做的任何事情“指针”也都能够做，为什么还要“引用”这东西？

答案是“用适当的工具做恰如其分的工作”。

指针能够毫无约束地操作内存中的如何东西，尽管指针功能强大，但是非常危险。就象一把刀，它可以用来砍树、裁纸、修指甲、理发等等，谁敢这样用？

如果的确只需要借用一下某个对象的“别名”，那么就用“引用”，而不要用“指针”，以免发生意外。比如说，某人需要一份证明，本来在文件上盖上公章的印子就行了，如果把取公章的钥匙交给他，那么他就获得了不该有的权利。

第 7 章 内存管理

未看

欢迎进入内存这片雷区。伟大的 Bill Gates 曾经失言：

640K ought to be enough for everybody

— Bill Gates 1981

程序员们经常编写内存管理程序，往往提心吊胆。如果不想触雷，唯一的解决办法就是发现所有潜伏的地雷并且排除它们，躲是躲不了的。本章的内容比一般教科书的要深入得多，读者需细心阅读，做到真正地通晓内存管理。

7.1 内存分配方式

内存分配方式有三种：

（1） 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，**static** 变量。

（2） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3） 从堆上分配，亦称动态内存分配。程序在运行的时候用 **malloc** 或 **new** 申请任意多少的内存，程序员自己负责在何时用 **free** 或 **delete** 释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

7.2 常见的内存错误及其对策

发生内存错误是件非常麻烦的事情。编译器不能自动发现这些错误，通常是在程序运行时才能捕捉到。而这些错误大多没有明显的症状，时隐时现，增加了改错的难度。有时用户怒气冲冲地把你找来，程序却没有发生任何问题，你一走，错误又发作了。

常见的内存错误及其对策如下：

◆ 内存分配未成功，却使用了它。

编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。常用解决办法是，在使用内存之前检查指针是否为 **NULL**。如果指针 **p** 是函数的参数，那么在函数的入口处用 `assert(p!=NULL)`进行检查。如果是用 `malloc` 或 `new` 来申请内存，应该用 `if(p==NULL)` 或 `if(p!=NULL)`进行防错处理。

◆ 内存分配虽然成功，但是尚未初始化就引用它。

犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。

内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可信

其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。

◆ 内存分配成功并且已经初始化，但操作越过了内存的边界。

例如在使用数组时经常发生下标“多 1”或者“少 1”的操作。特别是在 for 循环语句中，循环次数很容易搞错，导致数组操作越界。

◆ 忘记了释放内存，造成内存泄露。

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然死掉，系统出现提示：内存耗尽。

动态内存的申请与释放必须配对，程序中 malloc 与 free 的使用次数一定要相同，否则肯定有错误（new/delete 同理）。

◆ 释放了内存却继续使用它。

有三种情况：

（1）程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。

（2）函数的 return 语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。

（3）使用 free 或 delete 释放了内存后，没有将指针设置为 NULL。导致产生“野指针”。

- 【规则 7-2-1】用 malloc 或 new 申请内存之后，应该立即检查指针值是否为 NULL。防止使用指针值为 NULL 的内存。
- 【规则 7-2-2】不要忘记为数组和动态内存赋初值。防止将未被初始化的内存作为右值使用。
- 【规则 7-2-3】避免数组或指针的下标越界，特别要当心发生“多 1”或者“少 1”操作。
- 【规则 7-2-4】动态内存的申请与释放必须配对，防止内存泄漏。
- 【规则 7-2-5】用 free 或 delete 释放了内存之后，立即将指针设置为 NULL，防止产生“野指针”。

7.3 指针与数组的对比

C++/C 程序中，指针和数组在不少地方可以相互替换着用，让人产生一种错觉，以为两者是等价的。

数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。数组名对应着（而不是指向）一块内存，其地址与容量在生命期内保持不变，只有数组的内容可以改变。

指针可以随时指向任意类型的内存块，它的特征是“可变”，所以我们常用指针来

操作动态内存。指针远比数组灵活，但也更危险。

下面以字符串为例比较指针与数组的特性。

7.3.1 修改内容

示例 **7-3-1** 中，字符数组 **a** 的容量是 **6** 个字符，其内容为 **hello\0**。**a** 的内容可以改变，如 **a[0]= 'X'**。指针 **p** 指向常量字符串"**world**"（位于静态存储区，内容为 **world\0**），常量字符串的内容是不可以被修改的。从语法上看，编译器并不觉得语句 **p[0]= 'X'**有什么不妥，但是该语句企图修改常量字符串的内容而导致运行错误。

```
char a[] = "hello";
a[0] = 'X';
cout << a << endl;
char *p = "world";     // 注意 p 指向常量字符串
p[0] = 'X';            // 编译器不能发现该错误
cout << p << endl;
```

示例 **7-3-1** 修改数组和指针的内容

7.3.2 内容复制与比较

不能对数组名进行直接复制与比较。示例 7-3-2 中，若想把数组 a 的内容复制给数组 b，不能用语句 b = a ，否则将产生编译错误。应该用标准库函数 strcpy 进行复制。同理，比较 b 和 a 的内容是否相同，不能用 if(b==a) 来判断，应该用标准库函数 strcmp 进行比较。

语句 p = a 并不能把 a 的内容复制指针 p，而是把 a 的地址赋给了 p。要想复制 a 的内容，可以先用库函数 malloc 为 p 申请一块容量为 strlen(a)+1 个字符的内存，再用 strcpy 进行字符串复制。同理，语句 if(p==a) 比较的不是内容而是地址，应该用库函数 strcmp 来比较。

```
// 数组…
char a[] = "hello";
char b[10];
strcpy(b, a);          // 不能用   b = a;
if(strcmp(b, a) == 0)  // 不能用  if (b == a)
…
```

```
// 指针…
int len = strlen(a);
char *p = (char *)malloc(sizeof(char)*(len+1));
strcpy(p,a);           // 不要用 p = a;
if(strcmp(p, a) == 0)  // 不要用 if (p == a)
…
```

示例 **7-3-2** 数组和指针的内容复制与比较

7.3.3 计算内存容量

用运算符 sizeof 可以计算出数组的容量（字节数）。示例 7-3-3（a）中，sizeof(a) 的值是 12（注意别忘了'\0'）。指针 p 指向 a，但是 sizeof(p)的值却是 4。这是因为 sizeof(p)得到的是一个指针变量的字节数，相当于 sizeof(char*)，而不是 p 所指的内存容量。C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。

注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。示例 7-3-3（b）中，不论数组 a 的容量是多少，sizeof(a)始终等于 sizeof(char *)。

```
char a[] = "hello world";
char *p  = a;
cout<< sizeof(a) << endl; // 12 字节
cout<< sizeof(p) << endl; // 4 字节
```

示例 7-3-3（a） 计算数组和指针的内存容量

```
void Func(char a[100])
{
    cout<< sizeof(a) << endl; // 4 字节而不是 100 字节
}
```

示例 7-3-3（b） 数组退化为指针

7.4 指针参数是如何传递内存的？

如果函数的参数是一个指针，不要指望用该指针去申请动态内存。示例 **7-4-1** 中，Test 函数的语句 GetMemory(str, 200)并没有使 str 获得期望的内存，str 依旧是 NULL，为什么？

```
void GetMemory(char *p, int num)
{
    p = (char *)malloc(sizeof(char) * num);
}
```

```
void Test(void)
{
    char *str = NULL;
    GetMemory(str, 100);  // str 仍然为 NULL
    strcpy(str, "hello"); // 运行错误
}
```

示例 7-4-1 试图用指针参数申请动态内存

毛病出在函数 GetMemory 中。编译器总是要为函数的每个参数制作临时副本，指针参数 p 的副本是 _p，编译器使 _p = p。如果函数体内的程序修改了_p 的内容，就导致参数 p 的内容作相应的修改。这就是指针可以用作输出参数的原因。在本例中，_p 申请了新的内存，只是把_p 所指的内存地址改变了，但是 p 丝毫未变。所以函数 GetMemory 并不能输出任何东西。事实上，每执行一次 GetMemory 就会泄露一块内存，因为没有用 free 释放内存。

如果非得要用指针参数去申请内存，那么应该改用"指向指针的指针"，见示例 7-4-2。

```
void GetMemory2(char **p, int num)
{
    *p = (char *)malloc(sizeof(char) * num);
}
```

```
void Test2(void)
{
    char *str = NULL;
    GetMemory2(&str, 100); // 注意参数是 &str，而不是 str
    strcpy(str, "hello");
    cout<< str << endl;
    free(str);
}
```

示例 7-4-2 用指向指针的指针申请动态内存

由于"指向指针的指针"这个概念不容易理解，我们可以用函数返回值来传递动态内存。这种方法更加简单，见示例 7-4-3。

```
char *GetMemory3(int num)
{
    char *p = (char *)malloc(sizeof(char) * num);
    return p;
}
```

```
void Test3(void)
{
    char *str = NULL;
    str = GetMemory3(100);
    strcpy(str, "hello");
    cout<< str << endl;
    free(str);
}
```

示例 7-4-3 用函数返回值来传递动态内存

用函数返回值来传递动态内存这种方法虽然好用，但是常常有人把 **return** 语句用错了。这里强调不要用 **return** 语句返回指向"栈内存"的指针，因为该内存在函数结束时自动消亡，见示例 7-4-4。

```
char *GetString(void)
{
    char p[] = "hello world";
    return p;  // 编译器将提出警告
}
```

```
void Test4(void)
{
    char *str = NULL;
    str = GetString(); // str 的内容是垃圾
    cout<< str << endl;
}
```

示例 7-4-4 **return** 语句返回指向"栈内存"的指针

用调试器逐步跟踪 Test4，发现执行 str = GetString 语句后 str 不再是 NULL 指针，但是 str 的内容不是"hello world"而是垃圾。

如果把示例 7-4-4 改写成示例 7-4-5，会怎么样？

```
char *GetString2(void)
{
    char *p = "hello world";
    return p;
}
```

```
void Test5(void)
{
    char *str = NULL;
    str = GetString2();
    cout<< str << endl;
}
```

示例 7-4-5 **return** 语句返回常量字符串

函数 Test5 运行虽然不会出错，但是函数 GetString2 的设计概念却是错误的。因为 GetString2 内的"hello world"是常量字符串，位于静态存储区，它在程序生命期内恒定不变。无论什么时候调用 GetString2，它返回的始终是同一个"只读"的内存块。

7.5 free 和 delete 把指针怎么啦？

别看 **free** 和 **delete** 的名字恶狠狠的（尤其是 **delete**），它们只是把指针所指的内存给释放掉，但并没有把指针本身干掉。

用调试器跟踪示例 7-5，发现指针 **p** 被 **free** 以后其地址仍然不变（非 **NULL**），只是该地址对应的内存是垃圾，**p** 成了"野指针"。如果此时不把 **p** 设置为 **NULL**，会让人误以为 **p** 是个合法的指针。

如果程序比较长，我们有时记不住 **p** 所指的内存是否已经被释放，在继续使用 **p** 之前，通常会用语句 if (p != NULL)进行防错处理。很遗憾，此时 **if** 语句起不到防错作用，因为即便 **p** 不是 **NULL** 指针，它也不指向合法的内存块。

```
char *p = (char *) malloc(100);
strcpy(p, “hello”);
free(p);        // p 所指的内存被释放，但是 p 所指的地址仍然不变
…
if(p != NULL)  // 没有起到防错作用
{
   strcpy(p, “world”); // 出错
}
```

示例 7-5　p 成为野指针

7.6 动态内存会被自动释放吗？

函数体内的局部变量在函数结束时自动消亡。很多人误以为示例 7-6 是正确的。理由是 **p** 是局部的指针变量，它消亡的时候会让它所指的动态内存一起完蛋。这是错觉！

```
void Func(void)
{
    char *p = (char *) malloc(100);   // 动态内存会自动释放吗？
}
```

示例 7-6 试图让动态内存自动释放

我们发现指针有一些"似是而非"的特征：

（1）指针消亡了，并不表示它所指的内存会被自动释放。

（2）内存被释放了，并不表示指针会消亡或者成了 **NULL** 指针。

这表明释放内存并不是一件可以草率对待的事。也许有人不服气，一定要找出可以草率行事的理由：

如果程序终止了运行，一切指针都会消亡，动态内存会被操作系统回收。既然如此，

在程序临终前，就可以不必释放内存、不必将指针设置为 NULL 了。终于可以偷懒而不会发生错误了吧？

想得美。如果别人把那段程序取出来用到其它地方怎么办？

7.7 杜绝“野指针”

“野指针”不是 NULL 指针，是指向“垃圾”内存的指针。人们一般不会错用 NULL 指针，因为用 if 语句很容易判断。但是“野指针”是很危险的，if 语句对它不起作用。

“野指针”的成因主要有两种：

（1）指针变量没有被初始化。任何指针变量刚被创建时不会自动成为 NULL 指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为 NULL，要么让它指向合法的内存。例如

```
char *p = NULL;
char *str = (char *) malloc(100);
```

（2）指针 p 被 free 或者 delete 之后，没有置为 NULL，让人误以为 p 是个合法的指针。参见 7.5 节。

（3）指针操作超越了变量的作用范围。这种情况让人防不胜防，示例程序如下：

```
class A
{
public:
    void Func(void){ cout << “Func of class A” << endl; }
};
void Test(void)
{
    A  *p;
        {
            A  a;
            p = &a; // 注意 a 的生命期
        }
    p->Func();      // p是“野指针”
}
```

函数 Test 在执行语句 p->Func()时，对象 a 已经消失，而 p 是指向 a 的，所以 p 就成了“野指针”。但奇怪的是我运行这个程序时居然没有出错，这可能与编译器有关。

7.8 有了 malloc/free 为什么还要 new/delete ？

malloc 与 **free** 是 **C++/C** 语言的标准库函数，**new/delete** 是 **C++**的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，光用 **maloc/free** 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 **malloc/free** 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 **malloc/free**。

因此 **C++**语言需要一个能完成动态内存分配和初始化工作的运算符 **new**，以及一个能完成清理与释放内存工作的运算符 **delete**。注意 **new/delete** 不是库函数。

我们先看一看 **malloc/free** 和 **new/delete** 如何实现对象的动态内存管理，见示例 **7-8**。

```
class Obj
{
public :
        Obj(void){ cout << “Initialization” << endl; }
        ~Obj(void){ cout << “Destroy” << endl; }
void    Initialize(void){ cout << “Initialization” << endl; }
void     Destroy(void){ cout << “Destroy” << endl; }
};
```

```
void UseMallocFree(void)
{
    Obj  *a = (obj *)malloc(sizeof(obj));// 申请动态内存
    a->Initialize();                     // 初始化
    //…
    a->Destroy();  // 清除工作
    free(a);       // 释放内存
}
```

```
void UseNewDelete(void)
{
    Obj  *a = new Obj; // 申请动态内存并且初始化
    //…
    delete a;          // 清除并且释放内存
}
```

示例 **7-8** 用 **malloc/free** 和 **new/delete** 如何实现对象的动态内存管理

类 **Obj** 的函数 **Initialize** 模拟了构造函数的功能，函数 **Destroy** 模拟了析构函数的功能。函数 **UseMallocFree** 中，由于 **malloc/free** 不能执行构造函数与析构函数，必须调用成员函数 **Initialize** 和 **Destroy** 来完成初始化与清除工作。函数 **UseNewDelete** 则简单得

多。

所以我们不要企图用 **malloc/free** 来完成动态对象的内存管理，应该用 **new/delete**。由于内部数据类型的“对象”没有构造与析构的过程，对它们而言 **malloc/free** 和 **new/delete** 是等价的。

既然 new/delete 的功能完全覆盖了 malloc/free，为什么 C++不把 malloc/free 淘汰出局呢？这是因为 C++程序经常要调用 C 函数，而 C 程序只能用 malloc/free 管理动态内存。

如果用 free 释放“new 创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用 delete 释放“malloc 申请的动态内存”，理论上讲程序不会出错，但是该程序的可读性很差。所以 new/delete 必须配对使用，malloc/free 也一样。

7.9 内存耗尽怎么办？

如果在申请动态内存时找不到足够大的内存块，**malloc** 和 **new** 将返回 **NULL** 指针，宣告内存申请失败。通常有三种方式处理“内存耗尽”问题。

（**1**）判断指针是否为 **NULL**，如果是则马上用 **return** 语句终止本函数。例如：

```
void Func(void)
{
    A  *a = new A;
    if(a == NULL)
    {
        return;
    }
    …
}
```

（**2**）判断指针是否为 **NULL**，如果是则马上用 exit(1)终止整个程序的运行。例如：

```
void Func(void)
{
    A  *a = new A;
    if(a == NULL)
    {
        cout << “Memory Exhausted” << endl;
        exit(1);
    }
    …
}
```

（**3**）为 **new** 和 **malloc** 设置异常处理函数。例如 **Visual C++**可以用**_set_new_hander** 函

数为 **new** 设置用户自己定义的异常处理函数，也可以让 **malloc** 享用与 **new** 相同的异常处理函数。详细内容请参考 **C++**使用手册。

上述（**1**）（**2**）方式使用最普遍。如果一个函数内有多处需要申请动态内存，那么方式（**1**）就显得力不从心（释放内存很麻烦），应该用方式（**2**）来处理。

很多人不忍心用 **exit(1)**，问："不编写出错处理程序，让操作系统自己解决行不行？"

不行。如果发生"内存耗尽"这样的事情，一般说来应用程序已经无药可救。如果不用 **exit(1)** 把坏程序杀死，它可能会害死操作系统。道理如同：如果不把歹徒击毙，歹徒在老死之前会犯下更多的罪。

有一个很重要的现象要告诉大家。对于 **32** 位以上的应用程序而言，无论怎样使用 **malloc** 与 **new**，几乎不可能导致"内存耗尽"。我在 **Windows 98** 下用 **Visual C++**编写了测试程序，见示例 **7-9**。这个程序会无休止地运行下去，根本不会终止。因为 **32** 位操作系统支持"虚存"，内存用完了，自动用硬盘空间顶替。我只听到硬盘嘎吱嘎吱地响，**Window 98** 已经累得对键盘、鼠标毫无反应。

我可以得出这么一个结论：对于 **32** 位以上的应用程序，"内存耗尽"错误处理程序毫无用处。这下可把 **Unix** 和 **Windows** 程序员们乐坏了：反正错误处理程序不起作用，我就不写了，省了很多麻烦。

我不想误导读者，必须强调：不加错误处理将导致程序的质量很差，千万不可因小失大。

```
void main(void)
{
    float *p = NULL;
    while(TRUE)
    {
        p = new float[1000000];
        cout << “eat memory” << endl;
        if(p==NULL)
            exit(1);
    }
}
```

示例 **7-9** 试图耗尽操作系统的内存

7.10 malloc/free 的使用要点

函数 malloc 的原型如下：

```
void * malloc(size_t size);
```

用 malloc 申请一块长度为 length 的整数类型的内存，程序如下：

```
int  *p = (int *) malloc(sizeof(int) * length);
```

我们应当把注意力集中在两个要素上：“类型转换”和“sizeof”。

- malloc 返回值的类型是 void *，所以在调用 malloc 时要显式地进行类型转换，将 void * 转换成所需要的指针类型。
- malloc 函数本身并不识别要申请的内存是什么类型，它只关心内存的总字节数。我们通常记不住 int, float 等数据类型的变量的确切字节数。例如 int 变量在 16 位系统下是 2 个字节，在 32 位下是 4 个字节；而 float 变量在 16 位系统下是 4 个字节，在 32 位下也是 4 个字节。最好用以下程序作一次测试：

```
cout << sizeof(char) << endl;
cout << sizeof(int) << endl;
cout << sizeof(unsigned int) << endl;
cout << sizeof(long) << endl;
cout << sizeof(unsigned long) << endl;
cout << sizeof(float) << endl;
cout << sizeof(double) << endl;
cout << sizeof(void *) << endl;
```

在 malloc 的“()”中使用 sizeof 运算符是良好的风格，但要当心有时我们会昏了头，写出 p = malloc(sizeof(p))这样的程序来。

- 函数 free 的原型如下：

```
void free( void * memblock );
```

为什么 free 函数不象 malloc 函数那样复杂呢？这是因为指针 p 的类型以及它所指的内存的容量事先都是知道的，语句 free(p)能正确地释放内存。如果 p 是 NULL 指针，那么 free 对 p 无论操作多少次都不会出问题。如果 p 不是 NULL 指针，那么 free 对 p 连续操作两次就会导致程序运行错误。

7.11 new/delete 的使用要点

运算符 new 使用起来要比函数 malloc 简单得多，例如：

```
int  *p1 = (int *)malloc(sizeof(int) * length);
int  *p2 = new int[length];
```

这是因为 new 内置了 sizeof、类型转换和类型安全检查功能。对于非内部数据类型的对象而言，new 在创建动态对象的同时完成了初始化工作。如果对象有多个构造函数，那么 new 的语句也可以有多种形式。例如

```
class Obj
{
public :
    Obj(void);      // 无参数的构造函数
```

```
        Obj(int x);     // 带一个参数的构造函数
    …
    }
    void Test(void)
    {
        Obj  *a = new Obj;
        Obj  *b = new Obj(1); // 初值为 1
        …
        delete a;
        delete b;
    }
```

如果用 new 创建对象数组，那么只能使用对象的无参数构造函数。例如

```
    Obj  *objects = new Obj[100]; // 创建 100 个动态对象
```

不能写成

```
    Obj  *objects = new Obj[100](1);// 创建 100 个动态对象的同时赋初值 1
```

在用 delete 释放对象数组时，留意不要丢了符号‘[]’。例如

```
    delete []objects;  // 正确的用法
    delete objects;    // 错误的用法
```

后者相当于 delete objects[0]，漏掉了另外 99 个对象。

7.12 一些心得体会

我认识不少技术不错的 **C++/C** 程序员，很少有人能拍拍胸脯说通晓指针与内存管理（包括我自己）。我最初学习 **C** 语言时特别怕指针，导致我开发第一个应用软件（约 **1** 万行 **C** 代码）时没有使用一个指针，全用数组来顶替指针，实在蠢笨得过分。躲避指针不是办法，后来我改写了这个软件，代码量缩小到原先的一半。

我的经验教训是：

（**1**）越是怕指针，就越要使用指针。不会正确使用指针，肯定算不上是合格的程序员。

（**2**）必须养成“使用调试器逐步跟踪程序”的习惯，只有这样才能发现问题的本质。

第 8 章 C++函数的高级特性

对比于 C 语言的函数，C++增加了重载（overloaded）、内联（inline）、const 和 virtual 四种新机制。其中重载和内联机制既可用于全局函数也可用于类的成员函数，const 与 virtual 机制仅用于类的成员函数。

重载和内联肯定有其好处才会被 C++语言采纳，但是不可以当成免费的午餐而滥用。本章将探究重载和内联的优点与局限性，说明什么情况下应该采用、不该采用以及要警惕错用。

8.1 函数重载的概念

8.1.1 重载的起源

自然语言中，一个词可以有许多不同的含义，即该词被重载了。人们可以通过上下文来判断该词到底是哪种含义。“ 词的重载 ”可以使语言更加简练。例如“ 吃饭 ”的含义十分广泛，人们没有必要每次非得说清楚具体吃什么不可。别迂腐得象孔已己，说茴香豆的茴字有四种写法。

在 C++程序中，可以将语义、功能相似的几个函数用同一个名字表示，即函数重载。这样便于记忆，提高了函数的易用性，这是 C++语言采用重载机制的一个理由。例如示例 8-1-1 中的函数 EatBeef,EatFish,EatChicken 可以用同一个函数名 Eat 表示，用不同类型的参数加以区别。

```
void EatBeef(…);        // 可以改为    void Eat(Beef …);
void EatFish(…);        // 可以改为    void Eat(Fish …);
void EatChicken(…);     // 可以改为    void Eat(Chicken …);
```

示例 8-1-1 重载函数 Eat

C++语言采用重载机制的另一个理由是：类的构造函数需要重载机制。因为 C++规定构造函数与类同名（请参见第 9 章），构造函数只能有一个名字。如果想用几种不同的方法创建对象该怎么办？别无选择，只能用重载机制来实现。所以类可以有多个同名的构造函数。

8.1.2 重载是如何实现的？

几个同名的重载函数仍然是不同的函数，它们是如何区分的呢？我们自然想到函数接口的两个要素：参数与返回值。

如果同名函数的参数不同（包括类型、顺序不同），那么容易区别出它们是不同的函数。

如果同名函数仅仅是返回值类型不同，有时可以区分，有时却不能。例如：

```
void Function(void);
int  Function (void);
```

不能靠返回值判断

上述两个函数，第一个没有返回值，第二个的返回值是 int 类型。如果这样调用函数：

```
int  x = Function ();
```

则可以判断出 Function 是第二个函数。问题是在 C++/C 程序中，我们可以忽略函数的返回值。在这种情况下，编译器和程序员都不知道哪个 Function 函数被调用。

所以只能靠参数而不能靠返回值类型的不同来区分重载函数。编译器根据参数为每个重载函数产生不同的内部标识符。例如编译器为示例 8-1-1 中的三个 Eat 函数产生象 _eat_beef、_eat_fish、_eat_chicken 之类的内部标识符（不同的编译器可能产生不同风格的内部标识符）。

如果 C++程序要调用已经被编译后的 C 函数，该怎么办？

假设某个 C 函数的声明如下：

```
void foo(int x, int y);
```

该函数被 C 编译器编译后在库中的名字为_foo，而 C++编译器则会产生像_foo_int_int 之类的名字用来支持函数重载和类型安全连接。由于编译后的名字不同，C++程序不能直接调用 C 函数。C++提供了一个 C 连接交换指定符号 extern“C”来解决这个问题。例如：

```
extern “C”
{
   void foo(int x, int y);
   … // 其它函数
}
```

或者写成

```
extern “C”
{
   #include “myheader.h”
   … // 其它 C 头文件
}
```

这就告诉 C++编译译器，函数 foo 是个 C 连接，应该到库中找名字_foo 而不是找_foo_int_int。C++编译器开发商已经对 C 标准库的头文件作了 extern“C”处理，所以我们可以用＃include 直接引用这些头文件。

注意并不是两个函数的名字相同就能构成重载。全局函数和类的成员函数同名不算重载，因为函数的作用域不同。例如：

```
void Print(…);    // 全局函数
class A
{…
    void Print(…);    // 成员函数
```

```
}
```

不论两个 Print 函数的参数是否不同，如果类的某个成员函数要调用全局函数 Print，为了与成员函数 Print 区别，全局函数被调用时应加‘::’标志。如

```
::Print(…);    // 表示 Print 是全局函数而非成员函数
```

8.1.3 当心隐式类型转换导致重载函数产生二义性

示例 8-1-3 中，第一个 output 函数的参数是 int 类型，第二个 output 函数的参数是 float 类型。由于数字本身没有类型，将数字当作参数时将自动进行类型转换（称为隐式类型转换）。语句 output(0.5)将产生编译错误，因为编译器不知道该将 0.5 转换成 int 还是 float 类型的参数。隐式类型转换在很多地方可以简化程序的书写，但是也可能留下隐患。

```
# include <iostream.h>
void output( int x);  // 函数声明
void output( float x);// 函数声明

void output( int x)
{
    cout << " output int " << x << endl ;
}

void output( float x)
{
    cout << " output float " << x << endl ;
}

void main(void)
{
    int   x = 1;
    float y = 1.0;
    output(x);          // output int 1
    output(y);          // output float 1
    output(1);          // output int 1
//  output(0.5);        // error! ambiguous call, 因为自动类型转换
    output(int(0.5));  // output int 0
    output(float(0.5));    // output float 0.5
}
```

示例 8-1-3 隐式类型转换导致重载函数产生二义性

8.2 成员函数的重载、覆盖与隐藏

成员函数的重载、覆盖（override）与隐藏很容易混淆，C++程序员必须要搞清楚概念，否则错误将防不胜防。

8.2.1 重载与覆盖

成员函数被重载的特征：

（1）相同的范围（在同一个类中）；

（2）函数名字相同；

（3）参数不同；

（4）virtual 关键字可有可无。

覆盖是指派生类函数覆盖基类函数，特征是：

重写

（1）不同的范围（分别位于派生类与基类）；

（2）函数名字相同；

（3）参数相同；

（4）基类函数必须有 virtual 关键字。

示例 8-2-1 中，函数 Base::f(int)与 Base::f(float)相互重载，而 Base::g(void)被 Derived::g(void)覆盖。

```
#include <iostream.h>
class Base
{
  public:
         void f(int x){ cout << "Base::f(int) " << x << endl; }
         void f(float x){ cout << "Base::f(float) " << x << endl; }
  virtual void g(void){ cout << "Base::g(void)" << endl;}
};
```

```
class Derived : public Base
{
  public:
  virtual void g(void){ cout << "Derived::g(void)" << endl;}
};
```

```
void main(void)
{
  Derived  d;
  Base *pb = &d;
  pb->f(42);       // Base::f(int) 42
```

```
    pb->f(3.14f);    // Base::f(float) 3.14
    pb->g();         // Derived::g(void)
  }
```

示例 8-2-1 成员函数的重载和覆盖

8.2.2 令人迷惑的隐藏规则

本来仅仅区别重载与覆盖并不算困难，但是 C++的隐藏规则使问题复杂性陡然增加。这里“隐藏”是指派生类的函数屏蔽了与其同名的基类函数，规则如下：

（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无 virtual 关键字，基类的函数将被隐藏（注意别与重载混淆）。

（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有 virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

示例程序 8-2-2（a）中：

（1）函数 Derived::f(float)覆盖了 Base::f(float)。

（2）函数 Derived::g(int)隐藏了 Base::g(float)，而不是重载。

（3）函数 Derived::h(float)隐藏了 Base::h(float)，而不是覆盖。

```
#include <iostream.h>
class Base
{
public:
virtual void f(float x){ cout << "Base::f(float) " << x << endl; }
        void g(float x){ cout << "Base::g(float) " << x << endl; }
        void h(float x){ cout << "Base::h(float) " << x << endl; }
};
class Derived : public Base
{
public:
virtual void f(float x){ cout << "Derived::f(float) " << x << endl; }
        void g(int x){ cout << "Derived::g(int) " << x << endl; }
        void h(float x){ cout << "Derived::h(float) " << x << endl; }
};
```

示例 8-2-2（a）成员函数的重载、覆盖和隐藏

据作者考察，很多 C++程序员没有意识到有“隐藏”这回事。由于认识不够深刻，“隐藏”的发生可谓神出鬼没，常常产生令人迷惑的结果。

示例 8-2-2（b）中，bp 和 dp 指向同一地址，按理说运行结果应该是相同的，可事实并非这样。

```
void main(void)
{
  Derived  d;
  Base *pb = &d;
  Derived *pd = &d;
  // Good : behavior depends solely on type of the object
  pb->f(3.14f);   // Derived::f(float) 3.14
  pd->f(3.14f);   // Derived::f(float) 3.14

  // Bad : behavior depends on type of the pointer
  pb->g(3.14f);   // Base::g(float) 3.14
  pd->g(3.14f);   // Derived::g(int) 3        (surprise!)

  // Bad : behavior depends on type of the pointer
  pb->h(3.14f);   // Base::h(float) 3.14      (surprise!)
  pd->h(3.14f);   // Derived::h(float) 3.14
}
```

示例 8-2-2（b） 重载、覆盖和隐藏的比较

8.2.3 摆脱隐藏

隐藏规则引起了不少麻烦。示例 8-2-3 程序中，语句 pd->f(10)的本意是想调用函数 Base::f(int)，但是 Base::f(int)不幸被 Derived::f(char *)隐藏了。由于数字 10 不能被隐式地转化为字符串，所以在编译时出错。

```
class Base
{
  public:
    void f(int x);
};
```

```
class Derived : public Base
{
  public:
    void f(char *str);
};
```

```
void Test(void)
{
  Derived *pd = new Derived;
  pd->f(10);   // error
}
```

示例 8-2-3 由于隐藏而导致错误

从示例 8-2-3 看来，隐藏规则似乎很愚蠢。但是隐藏规则至少有两个存在的理由：

- ◆ 写语句 pd->f(10)的人可能真的想调用 Derived::f(char *)函数，只是他误将参数写错了。有了隐藏规则，编译器就可以明确指出错误，这未必不是好事。否则，编译器会静悄悄地将错就错，程序员将很难发现这个错误，流下祸根。
- ◆ 假如类 Derived 有多个基类（多重继承），有时搞不清楚哪些基类定义了函数 f。如果没有隐藏规则，那么 pd->f(10)可能会调用一个出乎意料的基类函数 f。尽管隐藏规则看起来不怎么有道理，但它的确能消灭这些意外。

示例 8-2-3 中，如果语句 pd->f(10)一定要调用函数 Base::f(int)，那么将类 Derived 修改为如下即可。

```
class Derived : public Base
{
  public:
  void f(char *str);
  void f(int x) { Base::f(x); }
};
```

8.3 参数的缺省值

有一些参数的值在每次函数调用时都相同，书写这样的语句会使人厌烦。C++语言采用参数的缺省值使书写变得简洁（在编译时，缺省值由编译器自动插入）。

参数缺省值的使用规则：

- 【规则 8-3-1】参数缺省值只能出现在函数的声明中，而不能出现在定义体中。

例如：

```
void Foo(int x=0, int y=0);   // 正确，缺省值出现在函数的声明中

void Foo(int x=0, int y=0)    // 错误，缺省值出现在函数的定义体中
{
…
}
```

为什么会这样？我想是有两个原因：一是函数的实现（定义）本来就与参数是否有缺省值无关，所以没有必要让缺省值出现在函数的定义体中。二是参数的缺省值可能会改动，显然修改函数的声明比修改函数的定义要方便。

- 【规则 8-3-2】如果函数有多个参数，参数只能从后向前挨个儿缺省，否则将导致函数调用语句怪模怪样。

正确的示例如下：

```
    void Foo(int x, int y=0, int z=0);
```

错误的示例如下：

```
void Foo(int x=0, int y, int z=0);
```

要注意，使用参数的缺省值并没有赋予函数新的功能，仅仅是使书写变得简洁一些。它可能会提高函数的易用性，但是也可能会降低函数的可理解性。所以我们只能适当地使用参数的缺省值，要防止使用不当产生负面效果。示例 8-3-2 中，不合理地使用参数的缺省值将导致重载函数 output 产生二义性。

```
#include <iostream.h>
void output( int x);
void output( int x, float y=0.0);
```

```
void output( int x)
{
    cout << " output int " << x << endl ;
}
```

```
void output( int x, float y)
{
    cout << " output int " << x << " and float " << y << endl ;
}
```

```
void main(void)
{
    int x=1;
    float y=0.5;
//  output(x);          // error! ambiguous call
    output(x,y);        // output int 1 and float 0.5
}
```

示例 8-3-2 参数的缺省值将导致重载函数产生二义性

8.4 运算符重载

8.4.1 概念

在 C++语言中，可以用关键字 operator 加上运算符来表示函数，叫做运算符重载。例如两个复数相加函数：

```
Complex Add(const Complex &a, const Complex &b);
```

可以用运算符重载来表示：

```
Complex operator +(const Complex &a, const Complex &b);
```

运算符与普通函数在调用时的不同之处是：对于普通函数，参数出现在圆括号内；而对于运算符，参数出现在其左、右侧。例如

```
Complex a, b, c;
…
c = Add(a, b); // 用普通函数
c = a + b;     // 用运算符 +
```

如果运算符被重载为全局函数，那么只有一个参数的运算符叫做一元运算符，有两个参数的运算符叫做二元运算符。

如果运算符被重载为类的成员函数，那么一元运算符没有参数，二元运算符只有一个右侧参数，因为对象自己成了左侧参数。

从语法上讲，运算符既可以定义为全局函数，也可以定义为成员函数。文献[Murray , p44-p47]对此问题作了较多的阐述，并总结了表 8-4-1 的规则。

运算符	规则
所有的一元运算符	建议重载为成员函数
= () [] ->	只能重载为成员函数
+= -= /= *= &= |= ~= %= >>= <<=	建议重载为成员函数
所有其它运算符	建议重载为全局函数

表 8-4-1 运算符的重载规则

由于 C++语言支持函数重载，才能将运算符当成函数来用，C 语言就不行。我们要以平常心来对待运算符重载：

（1）不要过分担心自己不会用，它的本质仍然是程序员们熟悉的函数。

（2）不要过分热心地使用，如果它不能使代码变得更加易读易写，那就别用，否则会自找麻烦。

8.4.2 不能被重载的运算符

在 C++运算符集合中，有一些运算符是不允许被重载的。这种限制是出于安全方面的考虑，可防止错误和混乱。

（1）不能改变 C++内部数据类型（如 int,float 等）的运算符。

（2）不能重载‘.’，因为‘.’在类中对任何成员都有意义，已经成为标准用法。

（3）不能重载目前 C++运算符集合中没有的符号，如#,@,$等。原因有两点，一是难以理解，二是难以确定优先级。

（4）对已经存在的运算符进行重载时，不能改变优先级规则，否则将引起混乱。

8.5 函数内联

8.5.1 用内联取代宏代码

C++ 语言支持函数内联，其目的是为了提高函数的执行效率（速度）。

在 C 程序中，可以用宏代码提高执行效率。宏代码本身不是函数，但使用起来象函数。预处理器用复制宏代码的方式代替函数调用，省去了参数压栈、生成汇编语言的 CALL 调用、返回参数、执行 return 等过程，从而提高了速度。使用宏代码最大的缺点是容易出错，预处理器在复制宏代码时常常产生意想不到的边际效应。例如

```
#define MAX(a, b)       (a) > (b) ? (a) : (b)
```

语句

```
result = MAX(i, j) + 2 ;
```

将被预处理器解释为

```
result = (i) > (j) ? (i) : (j) + 2 ;
```

由于运算符‘+’比运算符‘:’的优先级高，所以上述语句并不等价于期望的

```
result = ( (i) > (j) ? (i) : (j) ) + 2 ;
```

如果把宏代码改写为

```
#define MAX(a, b)       ( (a) > (b) ? (a) : (b) )
```

则可以解决由优先级引起的错误。但是即使使用修改后的宏代码也不是万无一失的，例如语句

```
result = MAX(i++, j);
```

将被预处理器解释为

```
result = (i++) > (j) ? (i++) : (j);
```

对于 C++ 而言，使用宏代码还有另一种缺点：无法操作类的私有数据成员。

让我们看看 C++ 的“函数内联”是如何工作的。对于任何内联函数，编译器在符号表里放入函数的声明（包括名字、参数类型、返回值类型）。如果编译器没有发现内联函数存在错误，那么该函数的代码也被放入符号表里。在调用一个内联函数时，编译器首先检查调用是否正确（进行类型安全检查，或者进行自动类型转换，当然对所有的函数都一样）。如果正确，内联函数的代码就会直接替换函数调用，于是省去了函数调用的开销。这个过程与预处理有显著的不同，因为预处理器不能进行类型安全检查，或者进行自动类型转换。假如内联函数是成员函数，对象的地址（this）会被放在合适的地方，这也是预处理器办不到的。

C++ 语言的函数内联机制既具备宏代码的效率，又增加了安全性，而且可以自由操作类的数据成员。所以在 C++ 程序中，应该用内联函数取代所有宏代码，“断言 assert”恐怕是唯一的例外。**assert** 是仅在 **Debug** 版本起作用的宏，它用于检查“不应该”发生的情况。为了不在程序的 **Debug** 版本和 **Release** 版本引起差别，**assert** 不应该产生任何副作用。如果 assert 是函数，由于函数调用会引起内存、代码的变动，那么将导致 Debug 版本与 Release 版本存在差异。所以 **assert** 不是函数，而是宏。（参见 **6.5** 节“使用断言”）

8.5.2 内联函数的编程风格

关键字 inline 必须与函数定义体放在一起才能使函数成为内联，仅将 inline 放在函数声明前面不起任何作用。如下风格的函数 Foo 不能成为内联函数：

```
inline void Foo(int x, int y);    // inline 仅与函数声明放在一起
```

```
void Foo(int x, int y)
{
    …
}
```

而如下风格的函数 Foo 则成为内联函数：

```
void Foo(int x, int y);
inline void Foo(int x, int y) // inline 与函数定义体放在一起
{
    …
}
```

所以说，inline 是一种“用于实现的关键字”，而不是一种“用于声明的关键字”。一般地，用户可以阅读函数的声明，但是看不到函数的定义。尽管在大多数教科书中内联函数的声明、定义体前面都加了 inline 关键字，但我认为 inline 不应该出现在函数的声明中。这个细节虽然不会影响函数的功能，但是体现了高质量 C++/C 程序设计风格的一个基本原则：声明与定义不可混为一谈，用户没有必要、也不应该知道函数是否需要内联。

定义在类声明之中的成员函数将自动地成为内联函数，例如

```
class A
{
public:
    void Foo(int x, int y) { … }     // 自动地成为内联函数
}
```

将成员函数的定义体放在类声明之中虽然能带来书写上的方便，但不是一种良好的编程风格，上例应该改成：

```
// 头文件
class A
{
public:
    void Foo(int x, int y)；
}
// 定义文件
inline void A::Foo(int x, int y)
{
    …
}
```

8.5.3 慎用内联

内联能提高函数的执行效率，为什么不把所有的函数都定义成内联函数？

如果所有的函数都是内联函数，还用得着“内联”这个关键字吗？

内联是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的

执行效率。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。以下情况不宜使用内联：

（1）如果函数体内的代码比较长，使用内联将导致内存消耗代价较高。

（2）如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。

类的构造函数和析构函数容易让人误解成使用内联更有效。要当心构造函数和析构函数可能会隐藏一些行为，如"偷偷地"执行了基类或成员对象的构造函数和析构函数。所以不要随便地将构造函数和析构函数的定义体放在类声明中。

一个好的编译器将会根据函数的定义体，自动地取消不值得的内联（这进一步说明了 inline 不应该出现在函数的声明中）。

8.6 一些心得体会

C++ 语言中的重载、内联、缺省参数、隐式转换等机制展现了很多优点，但是这些优点的背后都隐藏着一些隐患。正如人们的饮食，少食和暴食都不可取，应当恰到好处。我们要辨证地看待 C++的新机制，应该恰如其分地使用它们。虽然这会使我们编程时多费一些心思，少了一些痛快，但这才是编程的艺术。

第 9 章 类的构造函数、析构函数与赋值函数

构造函数、析构函数与赋值函数是每个类最基本的函数。它们太普通以致让人容易麻痹大意，其实这些貌似简单的函数就象没有顶盖的下水道那样危险。

每个类只有一个析构函数和一个赋值函数，但可以有多个构造函数（包含一个拷贝构造函数，其它的称为普通构造函数）。对于任意一个类 **A**，如果不想编写上述函数，**C++**编译器将自动为 **A** 产生四个缺省的函数，如

```
A(void);                    // 缺省的无参数构造函数
A(const A &a);              // 缺省的拷贝构造函数
~A(void);                   // 缺省的析构函数
A & operate =(const A &a);// 缺省的赋值函数
```

这不禁让人疑惑，既然能自动生成函数，为什么还要程序员编写？

原因如下：

（1）如果使用“缺省的无参数构造函数”和“缺省的析构函数”，等于放弃了自主“初始化”和“清除”的机会，C++发明人 Stroustrup 的好心好意白费了。

（2）“缺省的拷贝构造函数”和“缺省的赋值函数”均采用“位拷贝”而非“值拷贝”的方式来实现，倘若类中含有指针变量，这两个函数注定将出错。

对于那些没有吃够苦头的 **C++**程序员，如果他说编写构造函数、析构函数与赋值函数很容易，可以不用动脑筋，表明他的认识还比较肤浅，水平有待于提高。

本章以类 **String** 的设计与实现为例，深入阐述被很多教科书忽视了的道理。**String** 的结构如下：

```
class String
{
  public:
    String(const char *str = NULL);   // 普通构造函数
    String(const String &other);  // 拷贝构造函数
    ~ String(void);                   // 析构函数
    String & operate =(const String &other); // 赋值函数
  private:
    char    *m_data;               // 用于保存字符串
};
```

9.1 构造函数与析构函数的起源

作为比 **C** 更先进的语言，**C++**提供了更好的机制来增强程序的安全性。**C++**编译器具有严格的类型安全检查功能，它几乎能找出程序中所有的语法问题，这的确帮了程序

员的大忙。但是程序通过了编译检查并不表示错误已经不存在了，在“错误”的大家庭里，“语法错误”的地位只能算是小弟弟。级别高的错误通常隐藏得很深，就象狡猾的罪犯，想逮住他可不容易。

根据经验，不少难以察觉的程序错误是由于变量没有被正确初始化或清除造成的，而初始化和清除工作很容易被人遗忘。**Stroustrup** 在设计 **C++**语言时充分考虑了这个问题并很好地予以解决：把对象的初始化工作放在构造函数中，把清除工作放在析构函数中。当对象被创建时，构造函数被自动执行。当对象消亡时，析构函数被自动执行。这下就不用担心忘了对象的初始化和清除工作。

构造函数与析构函数的名字不能随便起，必须让编译器认得出才可以被自动执行。**Stroustrup** 的命名方法既简单又合理：让构造函数、析构函数与类同名，由于析构函数的目的与构造函数的相反，就加前缀‘~’以示区别。

除了名字外，构造函数与析构函数的另一个特别之处是没有返回值类型，这与返回值类型为 **void** 的函数不同。构造函数与析构函数的使命非常明确，就象出生与死亡，光溜溜地来光溜溜地去。如果它们有返回值类型，那么编译器将不知所措。为了防止节外生枝，干脆规定没有返回值类型。（以上典故参考了文献**[Eekel, p55-p56]**）

9.2 构造函数的初始化表

构造函数有个特殊的初始化方式叫“初始化表达式表”（简称初始化表）。初始化表位于函数参数表之后，却在函数体 **{}** 之前。这说明该表里的初始化工作发生在函数体内的任何代码被执行之前。

构造函数初始化表的使用规则：

◆ 如果类存在继承关系，派生类必须在其初始化表里调用基类的构造函数。

例如

```
class A
{…
    A(int x);       // A 的构造函数
};
class B : public A
{…
    B(int x, int y);// B 的构造函数
};
B::B(int x, int y)
 : A(x)             // 在初始化表里调用 A 的构造函数
{
  …
}
```

◆ 类的 **const** 常量只能在初始化表里被初始化，因为它不能在函数体内用赋值的方式来初始化（参见 5.4 节）。

◆ 类的数据成员的初始化可以采用初始化表或函数体内赋值两种方式，这两种方式的效率不完全相同。

非内部数据类型的成员对象应当采用第一种方式初始化，以获取更高的效率。例如

```
class A
{…
    A(void);                // 无参数构造函数
    A(const A &other);      // 拷贝构造函数
    A & operate =( const A &other);   // 赋值函数
}；

class B
{
  public:
    B(const A &a); // B 的构造函数
  private:
    A  m_a;         // 成员对象
};
```

示例 9-2(a)中，类 B 的构造函数在其初始化表里调用了类 A 的拷贝构造函数，从而将成员对象 m_a 初始化。

示例 9-2 (b)中，类 B 的构造函数在函数体内用赋值的方式将成员对象 m_a 初始化。我们看到的只是一条赋值语句，但实际上 B 的构造函数干了两件事：先暗地里创建 m_a 对象（调用了 A 的无参数构造函数），再调用类 A 的赋值函数，将参数 a 赋给 m_a。

B::B(const A &a)
 : m_a(a)
{
   …
}
	B::B(const A &a)
	{
	    m_a = a;
	    …
	}

示例 9-2(a) 成员对象在初始化表中被初始化	示例 9-2(b) 成员对象在函数体内被初始化

对于内部数据类型的数据成员而言，两种初始化方式的效率几乎没有区别，但后者的程序版式似乎更清晰些。若类 F 的声明如下：

```
class F
{
  public:
    F(int x, int y);        // 构造函数
  private:
    int m_x, m_y;
    int m_i, m_j;
```

```
}
```

示例 9-2(c)中 F 的构造函数采用了第一种初始化方式，示例 9-2(d)中 F 的构造函数采用了第二种初始化方式。

示例 9-2(c) 数据成员在初始化表中被初始化	示例 9-2(d) 数据成员在函数体内被初始化

F::F(int x, int y)
 : m_x(x), m_y(y)
{
   m_i = 0;
   m_j = 0;
}
	F::F(int x, int y)
	{
	   m_x = x;
	   m_y = y;
	   m_i = 0;
	   m_j = 0;
	}

9.3 构造和析构的次序

构造从类层次的最根处开始，在每一层中，首先调用基类的构造函数，然后调用成员对象的构造函数。析构则严格按照与构造相反的次序执行，该次序是唯一的，否则编译器将无法自动执行析构过程。

一个有趣的现象是，成员对象初始化的次序完全不受它们在初始化表中次序的影响，只由成员对象在类中声明的次序决定。这是因为类的声明是唯一的，而类的构造函数可以有多个，因此会有多个不同次序的初始化表。如果成员对象按照初始化表的次序进行构造，这将导致析构函数无法得到唯一的逆序。**[Eckel, p260-261]**

9.4 示例：类 String 的构造函数与析构函数

```
// String 的普通构造函数
String::String(const char *str)
{
    if(str==NULL)
    {
        m_data = new char[1];
        *m_data = '\0';
    }
    else
    {
        int length = strlen(str);
        m_data = new char[length+1];
        strcpy(m_data, str);
```

```
    }
}

// String 的析构函数
String::~String(void)
{
    delete [] m_data;
    // 由于 m_data 是内部数据类型，也可以写成 delete m_data;
}
```

9.5 不要轻视拷贝构造函数与赋值函数

由于并非所有的对象都会使用拷贝构造函数和赋值函数，程序员可能对这两个函数有些轻视。请先记住以下的警告，在阅读正文时就会多心：

- 本章开头讲过，如果不主动编写拷贝构造函数和赋值函数，编译器将以“位拷贝”的方式自动生成缺省的函数。倘若类中含有指针变量，那么这两个缺省的函数就隐含了错误。以类 String 的两个对象 a,b 为例，假设 a.m_data 的内容为“hello”，b.m_data 的内容为“world”。

  现将 a 赋给 b，缺省赋值函数的“位拷贝”意味着执行 b.m_data = a.m_data。这将造成三个错误：一是 b.m_data 原有的内存没被释放，造成内存泄露；二是 b.m_data 和 a.m_data 指向同一块内存，a 或 b 任何一方变动都会影响另一方；三是在对象被析构时，m_data 被释放了两次。

- 拷贝构造函数和赋值函数非常容易混淆，常导致错写、错用。拷贝构造函数是在对象被创建时调用的，而赋值函数只能被已经存在了的对象调用。以下程序中，第三个语句和第四个语句很相似，你分得清楚哪个调用了拷贝构造函数，哪个调用了赋值函数吗？

```
    String  a(“hello”);
    String  b(“world”);
    String  c = a; // 调用了拷贝构造函数，最好写成 c(a);
            c = b; // 调用了赋值函数
```

  本例中第三个语句的风格较差，宜改写成 String c(a) 以区别于第四个语句。

9.6 示例：类 String 的拷贝构造函数与赋值函数

```
// 拷贝构造函数
String::String(const String &other)
{
```

```
        // 允许操作 other 的私有成员 m_data
        int length = strlen(other.m_data);
        m_data = new char[length+1];
        strcpy(m_data, other.m_data);
    }

    // 赋值函数
    String & String::operate =(const String &other)
    {
        // (1) 检查自赋值
        if(this == &other)
            return *this;

        // (2) 释放原有的内存资源
        delete [] m_data;

        // （3）分配新的内存资源，并复制内容
        int length = strlen(other.m_data);
        m_data = new char[length+1];
        strcpy(m_data, other.m_data);

        // （4）返回本对象的引用
        return *this;
    }
```

类 **String** 拷贝构造函数与普通构造函数（参见 **9.4** 节）的区别是：在函数入口处无需与 NULL 进行比较，这是因为“引用”不可能是 NULL，而“指针”可以为 NULL。

类 **String** 的赋值函数比构造函数复杂得多，分四步实现：

（**1**）第一步，检查自赋值。你可能会认为多此一举，难道有人会愚蠢到写出 a = a 这样的自赋值语句！的确不会。但是间接的自赋值仍有可能出现，例如

// 内容自赋值	// 地址自赋值

b = a;
…
c = b;
…
a = c;
	b = &a;
	…
	a = *b;

也许有人会说：“即使出现自赋值，我也可以不理睬，大不了化点时间让对象复制自己而已，反正不会出错！”

他真的说错了。看看第二步的 delete，自杀后还能复制自己吗？所以，如果发现自赋值，应该马上终止函数。注意不要将检查自赋值的 if 语句

```
if(this == &other)
```

错写成为

```
if( *this == other)
```

（2）第二步，用 delete 释放原有的内存资源。如果现在不释放，以后就没机会了，将造成内存泄露。

（3）第三步，分配新的内存资源，并复制字符串。注意函数 strlen 返回的是有效字符串长度，不包含结束符‘\0’。函数 strcpy 则连‘\0’一起复制。

（4）第四步，返回本对象的引用，目的是为了实现象 a = b = c 这样的链式表达。注意不要将 return *this 错写成 return this 。那么能否写成 return other 呢？效果不是一样吗？

不可以！因为我们不知道参数 other 的生命期。有可能 other 是个临时对象，在赋值结束后它马上消失，那么 return other 返回的将是垃圾。

9.7 偷懒的办法处理拷贝构造函数与赋值函数

如果我们实在不想编写拷贝构造函数和赋值函数，又不允许别人使用编译器生成的缺省函数，怎么办？

偷懒的办法是：只需将拷贝构造函数和赋值函数声明为私有函数，不用编写代码。

例如：

```
class A
{ …
  private:
    A(const A &a);              // 私有的拷贝构造函数
    A & operate =(const A &a);// 私有的赋值函数
};
```

如果有人试图编写如下程序：

```
A  b(a);    // 调用了私有的拷贝构造函数
b = a;      // 调用了私有的赋值函数
```

编译器将指出错误，因为外界不可以操作 **A** 的私有函数。

9.8 如何在派生类中实现类的基本函数

基类的构造函数、析构函数、赋值函数都不能被派生类继承。如果类之间存在继承关系，在编写上述基本函数时应注意以下事项：

- 派生类的构造函数应在其初始化表里调用基类的构造函数。

◆ 基类与派生类的析构函数应该为虚（即加 **virtual** 关键字）。例如

```
#include <iostream.h>
class Base
{
  public:
    virtual ~Base() { cout<< "~Base" << endl ; }
};

class Derived : public Base
{
  public:
    virtual ~Derived() { cout<< "~Derived" << endl ; }
};

void main(void)
{
    Base * pB = new Derived;  // upcast
    delete pB;
}
```

输出结果为：

~Derived

~Base

如果析构函数不为虚，那么输出结果为

~Base

◆ 在编写派生类的赋值函数时，注意不要忘记对基类的数据成员重新赋值。例如：

```
class Base
{
  public:
    …
    Base & operate =(const Base &other); // 类 Base 的赋值函数
  private:
    int  m_i, m_j, m_k;
};

class Derived : public Base
{
  public:
    …
```

```
    Derived & operate =(const Derived &other);   // 类 Derived 的赋值函数
  private:
    int  m_x, m_y, m_z;
};

Derived & Derived::operate =(const Derived &other)
{
    //（1）检查自赋值
    if(this == &other)
        return *this;

    //（2）对基类的数据成员重新赋值
    Base::operate =(other);   // 因为不能直接操作私有数据成员

    //（3）对派生类的数据成员赋值
    m_x = other.m_x;
    m_y = other.m_y;
    m_z = other.m_z;

    //（4）返回本对象的引用
    return *this;
}
```

9.9 一些心得体会

有些 C++程序设计书籍称构造函数、析构函数和赋值函数是类的“ Big-Three ”，它们的确是任何类最重要的函数，不容轻视。

也许你认为本章的内容已经够多了，学会了就能平安无事，我不能作这个保证。如果你希望吃透“ Big-Three ”，请好好阅读参考文献[Cline] [Meyers] [Murry]。

第 10 章 类的继承与组合

对象（Object）是类（Class）的一个实例（Instance）。如果将对象比作房子，那么类就是房子的设计图纸。所以面向对象设计的重点是类的设计，而不是对象的设计。

对于 C++程序而言，设计孤立的类是比较容易的，难的是正确设计基类及其派生类。本章仅仅论述“继承”（Inheritance）和“组合”（Composition）的概念。

注意，当前面向对象技术的应用热点是 COM 和 CORBA，这些内容超出了 C++教材的范畴，请阅读 COM 和 CORBA 相关论著。

10.1 继承

如果 A 是基类，B 是 A 的派生类，那么 B 将继承 A 的数据和函数。例如：

```
class A
{
  public:
        void   Func1(void);
        void   Func2(void);
};

class B : public A
{
  public:
        void   Func3(void);
        void   Func4(void);
};

main()
{
    B   b;
    b.Func1();         // B 从 A 继承了函数 Func1
    b.Func2();         // B 从 A 继承了函数 Func2
    b.Func3();
    b.Func4();
}
```

这个简单的示例程序说明了一个事实：C++的“继承”特性可以提高程序的可复用性。正因为“继承”太有用、太容易用，才要防止乱用“继承”。我们应当给“继承”

立一些使用规则。

- 【规则 10-1-1】如果类 A 和类 B 毫不相关，不可以为了使 B 的功能更多些而让 B 继承 A 的功能和属性。不要觉得“白吃白不吃”，让一个好端端的健壮青年无缘无故地吃人参补身体。
- 【规则 10-1-2】若在逻辑上 B 是 A 的“一种”（a kind of ），则允许 B 继承 A 的功能和属性。例如男人（Man）是人（Human）的一种，男孩（Boy）是男人的一种。那么类 Man 可以从类 Human 派生，类 Boy 可以从类 Man 派生。

```
class Human
{
      …
};
class Man : public Human
{
      …
};
class Boy : public Man
{
      …
};
```

◆ 注意事项

【规则 10-1-2】看起来很简单，但是实际应用时可能会有意外，继承的概念在程序世界与现实世界并不完全相同。

例如从生物学角度讲，鸵鸟（Ostrich）是鸟（Bird）的一种，按理说类 Ostrich 应该可以从类 Bird 派生。但是鸵鸟不能飞，那么 Ostrich::Fly 是什么东西？

```
class Bird
{
public:
    virtual void Fly(void);
…
};

class Ostrich : public Bird
{
…
};
```

例如从数学角度讲，圆（Circle）是一种特殊的椭圆（Ellipse），按理说类 Circle 应该可以从类 Ellipse 派生。但是椭圆有长轴和短轴，如果圆继承了椭圆的长轴和短轴，

岂非画蛇添足？

所以更加严格的继承规则应当是：若在逻辑上 B 是 A 的“一种”，并且 A 的所有功能和属性对 B 而言都有意义，则允许 B 继承 A 的功能和属性。

10.2 组合

- 【规则 10-2-1】若在逻辑上 A 是 B 的“一部分”（a part of），则不允许 B 从 A 派生，而是要用 A 和其它东西组合出 B。

例如眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类 Head 应该由类 Eye、Nose、Mouth、Ear 组合而成，不是派生而成。如示例 10-2-1 所示。

```
class Eye
{
  public:
    void   Look(void);
};
```

```
class Nose
{
  public:
    void   Smell(void);
};
```

```
class Mouth
{
  public:
    void   Eat(void);
};
```

```
class Ear
{
  public:
    void   Listen(void);
};
```

```
// 正确的设计，虽然代码冗长。
class Head
{
  public:
        void    Look(void)   {  m_eye.Look();  }
        void    Smell(void)  {  m_nose.Smell();  }
        void    Eat(void)    {  m_mouth.Eat();  }
        void    Listen(void) {  m_ear.Listen();  }
  private:
        Eye     m_eye;
        Nose    m_nose;
        Mouth       m_mouth;
        Ear     m_ear;
};
```

示例 10-2-1 Head 由 Eye、Nose、Mouth、Ear 组合而成

如果允许 Head 从 Eye、Nose、Mouth、Ear 派生而成，那么 Head 将自动具有 Look、

Smell、Eat、Listen 这些功能。示例 10-2-2 十分简短并且运行正确，但是这种设计方法却是不对的。

```
// 功能正确并且代码简洁，但是设计方法不对。
class Head : public Eye, public Nose, public Mouth, public Ear
{
};
```

示例 10-2-2　Head 从 Eye、Nose、Mouth、Ear 派生而成

一只公鸡使劲地追打一只刚下了蛋的母鸡，你知道为什么吗？

因为母鸡下了鸭蛋。

很多程序员经不起“继承”的诱惑而犯下设计错误。“运行正确”的程序不见得是高质量的程序，此处就是一个例证。

第 11 章 其它编程经验

11.1 使用 const 提高函数的健壮性

看到 const 关键字，C++程序员首先想到的可能是 const 常量。这可不是良好的条件反射。如果只知道用 const 定义常量，那么相当于把火药仅用于制作鞭炮。const 更大的魅力是它可以修饰函数的参数、返回值，甚至函数的定义体。

const 是 constant 的缩写，“恒定不变”的意思。被 const 修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。所以很多 C++程序设计书籍建议：“Use const whenever you need”。

11.1.1 用 const 修饰函数的参数

如果参数作输出用，不论它是什么数据类型，也不论它采用“指针传递”还是“引用传递”，都不能加 const 修饰，否则该参数将失去输出功能。

const 只能修饰输入参数：

◆ 如果输入参数采用“指针传递”，那么加 const 修饰可以防止意外地改动该指针，起到保护作用。

例如 StringCopy 函数：

```
void StringCopy(char *strDestination, const char *strSource);
```

其中 strSource 是输入参数，strDestination 是输出参数。给 strSource 加上 const 修饰后，如果函数体内的语句试图改动 strSource 的内容，编译器将指出错误。

◆ 如果输入参数采用“值传递”，由于函数将自动产生临时变量用于复制该参数，该输入参数本来就无需保护，所以不要加 const 修饰。

例如不要将函数 void Func1(int x) 写成 void Func1(const int x)。同理不要将函数 void Func2(A a) 写成 void Func2(const A a)。其中 A 为用户自定义的数据类型。

◆ 对于非内部数据类型的参数而言，象 void Func(A a) 这样声明的函数注定效率比较底。因为函数体内将产生 A 类型的临时对象用于复制参数 a，而临时对象的构造、复制、析构过程都将消耗时间。

为了提高效率，可以将函数声明改为 void Func(A &a)，因为“引用传递”仅借用一下参数的别名而已，不需要产生临时对象。但是函数 void Func(A &a) 存在一个缺点：“引用传递”有可能改变参数 a，这是我们不期望的。解决这个问题很容易，加 const 修饰即可，因此函数最终成为 void Func(const A &a)。

以此类推，是否应将 void Func(int x) 改写为 void Func(const int &x)，以便提高效率？完全没有必要，因为内部数据类型的参数不存在构造、析构的过程，而复制也非常快，“值传递”和“引用传递”的效率几乎相当。

问题是如此的缠绵，我只好将“const &”修饰输入参数的用法总结一下，如表 11-1-1

所示。

对于非内部数据类型的输入参数，应该将“值传递”的方式改为“const 引用传递”，目的是提高效率。例如将 void Func(A a) 改为 void Func(const A &a)。
对于内部数据类型的输入参数，不要将“值传递”的方式改为“const 引用传递”。否则既达不到提高效率的目的，又降低了函数的可理解性。例如 void Func(int x) 不应该改为 void Func(const int &x)。

表 11-1-1 “const &”修饰输入参数的规则

11.1.2 用 const 修饰函数的返回值

◆ 如果给以“指针传递”方式的函数返回值加 const 修饰，那么函数返回值（即指针）的内容不能被修改，该返回值只能被赋给加 const 修饰的同类型指针。

例如函数

```
const char * GetString(void);
```

如下语句将出现编译错误：

```
char *str = GetString();
```

正确的用法是

```
const char *str = GetString();
```

◆ 如果函数返回值采用“值传递方式”，由于函数会把返回值复制到外部临时的存储单元中，加 const 修饰没有任何价值。

例如不要把函数 int GetInt(void) 写成 const int GetInt(void)。

同理不要把函数 A GetA(void) 写成 const A GetA(void)，其中 A 为用户自定义的数据类型。

如果返回值不是内部数据类型，将函数 A GetA(void) 改写为 const A & GetA(void) 的确能提高效率。但此时千万千万要小心，一定要搞清楚函数究竟是想返回一个对象的“拷贝”还是仅返回“别名”就可以了，否则程序会出错。见 6.2 节“返回值的规则”。

◆ 函数返回值采用“引用传递”的场合并不多，这种方式一般只出现在类的赋值函数中，目的是为了实现链式表达。

例如

```
class A
{…
    A & operate = (const A &other);   // 赋值函数
};
A a, b, c;      // a, b, c 为 A 的对象
…
a = b = c;          // 正常的链式赋值
```

```
(a = b) = c;        // 不正常的链式赋值，但合法
```

如果将赋值函数的返回值加 const 修饰，那么该返回值的内容不允许被改动。上例中，语句 a = b = c 仍然正确，但是语句 (a = b) = c 则是非法的。

11.1.3 const 成员函数

任何不会修改数据成员的函数都应该声明为 const 类型。如果在编写 const 成员函数时，不慎修改了数据成员，或者调用了其它非 const 成员函数，编译器将指出错误，这无疑会提高程序的健壮性。

以下程序中，类 **stack** 的成员函数 **GetCount** 仅用于计数，从逻辑上讲 **GetCount** 应当为 **const** 函数。编译器将指出 **GetCount** 函数中的错误。

```
class Stack
{
  public:
    void   Push(int elem);
    int    Pop(void);
    int    GetCount(void)  const; // const 成员函数
  private:
    int    m_num;
    int    m_data[100];
};

int Stack::GetCount(void)  const
{
    ++ m_num;  // 编译错误，企图修改数据成员 m_num
    Pop();     // 编译错误，企图调用非 const 函数
    return m_num;
}
```

const 成员函数的声明看起来怪怪的：const 关键字只能放在函数声明的尾部，大概是因为其它地方都已经被占用了。

11.2 提高程序的效率

程序的时间效率是指运行速度，空间效率是指程序占用内存或者外存的状况。

全局效率是指站在整个系统的角度上考虑的效率，局部效率是指站在模块或函数角度上考虑的效率。

- 【**规则 11-2-1**】不要一味地追求程序的效率，应当在满足正确性、可靠性、健壮性、可读性等质量因素的前提下，设法提高程序的效率。

- 【规则 **11-2-2**】以提高程序的全局效率为主，提高局部效率为辅。

- 【规则 **11-2-3**】在优化程序的效率时，应当先找出限制效率的“瓶颈”，不要在无关紧要之处优化。

- 【规则 **11-2-4**】先优化数据结构和算法，再优化执行代码。

- 【规则 **11-2-5**】有时候时间效率和空间效率可能对立，此时应当分析那个更重要，作出适当的折衷。例如多花费一些内存来提高性能。

- 【规则 **11-2-6**】不要追求紧凑的代码，因为紧凑的代码并不能产生高效的机器码。

11.3 一些有益的建议

- ✧ 【建议 **11-3-1**】当心那些视觉上不易分辨的操作符发生书写错误。
  我们经常会把“＝＝”误写成“＝”，象“||”、“&&”、“<=”、“>=”这类符号也很容易发生“丢 1”失误。然而编译器却不一定能自动指出这类错误。

- ✧ 【建议 **11-3-2**】变量（指针、数组）被创建之后应当及时把它们初始化，以防止把未被初始化的变量当成右值使用。

- ✧ 【建议 **11-3-3**】当心变量的初值、缺省值错误，或者精度不够。

- ✧ 【建议 **11-3-4**】当心数据类型转换发生错误。尽量使用显式的数据类型转换（让人们知道发生了什么事），避免让编译器轻悄悄地进行隐式的数据类型转换。

- ✧ 【建议 **11-3-5**】当心变量发生上溢或下溢，数组的下标越界。

- ✧ 【建议 **11-3-6**】当心忘记编写错误处理程序，当心错误处理程序本身有误。

- ✧ 【建议 **11-3-7**】当心文件 **I/O** 有错误。

- ✧ 【建议 **11-3-8**】避免编写技巧性很高代码。

- ✧ 【建议 **11-3-9**】不要设计面面俱到、非常灵活的数据结构。

- ✧ 【建议 **11-3-10**】如果原有的代码质量比较好，尽量复用它。但是不要修补很差劲的代码，应当重新编写。

✧ **【建议 11-3-11】尽量使用标准库函数，不要“发明”已经存在的库函数。**

✧ **【建议 11-3-12】尽量不要使用与具体硬件或软件环境关系密切的变量。**

✧ **【建议 11-3-13】把编译器的选择项设置为最严格状态。**

✧ **【建议 11-3-14】如果可能的话，使用 PC-Lint、LogiScope 等工具进行代码审查。**

参考文献

[Cline] Marshall P. Cline and Greg A. Lomow, C++ FAQs, Addison-Wesley, 1995

[Eckel] Bruce Eckel, Thinking in C++（C++ 编程思想，刘宗田 等译），机械工业出版社，2000

[Maguire] Steve Maguire, Writing Clean Code（编程精粹，姜静波 等译），电子工业出版社，1993

[Meyers] Scott Meyers, Effective C++, Addison-Wesley, 1992

[Murry] Robert B. Murry, C++ Strategies and Tactics, Addison-Wesley, 1993

[Summit] Steve Summit, C Programming FAQs, Addison-Wesley, 1996

附录 A ：C++/C 代码审查表

文件结构		
重要性	审查项	结论
	头文件和定义文件的名称是否合理？	
	头文件和定义文件的目录结构是否合理？	
	版权和版本声明是否完整？	
重要	头文件是否使用了 ifndef/define/endif 预处理块?	
	头文件中是否只存放“声明”而不存放“定义”	
	……	
程序的版式		
重要性	审查项	结论
	空行是否得体？	
	代码行内的空格是否得体？	
	长行拆分是否得体？	
	“{” 和 “}” 是否各占一行并且对齐于同一列？	
重要	一行代码是否只做一件事？如只定义一个变量，只写一条语句。	
重要	If、for、while、do 等语句自占一行，不论执行语句多少都要加“{}”。	
重要	在定义变量（或参数）时，是否将修饰符 * 和 & 紧靠变量名？	
	注释是否清晰并且必要？	
重要	注释是否有错误或者可能导致误解？	
重要	类结构的 public，protected，private 顺序是否在所有的程序中保持一致？	
	……	
命名规则		
重要性	审查项	结论
重要	命名规则是否与所采用的操作系统或开发工具的风格保持一致？	
	标识符是否直观且可以拼读？	
	标识符的长度应当符合“min-length && max-information”原则？	
重要	程序中是否出现相同的局部变量和全部变量？	
	类名、函数名、变量和参数、常量的书写格式是否遵循一定的规则？	
	静态变量、全局变量、类的成员变量是否加前缀？	

	……	
表达式与基本语句		
重要性	审查项	结论
重要	如果代码行中的运算符比较多，是否已经用括号清楚地确定表达式的操作顺序？	
	是否编写太复杂或者多用途的复合表达式？	
重要	是否将复合表达式与“真正的数学表达式”混淆？	
重要	是否用隐含错误的方式写 if 语句？例如 （1）将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较。 （2）将浮点变量用“==”或“!=”与任何数字比较。 （3）将指针变量用“==”或“!=”与 NULL 比较。	
	如果循环体内存在逻辑判断，并且循环次数很大，是否已经将逻辑判断移到循环体的外面？	
重要	Case 语句的结尾是否忘了加 break？	
重要	是否忘记写 switch 的 default 分支？	
重要	使用 goto 语句时是否留下隐患？例如跳过了某些对象的构造、变量的初始化、重要的计算等。	
	……	
常量		
重要性	审查项	结论
	是否使用含义直观的常量来表示那些将在程序中多次出现的数字或字符串？	
	在 C++ 程序中，是否用 const 常量取代宏常量？	
重要	如果某一常量与其它常量密切相关，是否在定义中包含了这种关系？	
	是否误解了类中的 const 数据成员？因为 const 数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的。	
	……	
函数设计		
重要性	审查项	结论
	参数的书写是否完整？不要贪图省事只写参数的类型而省略参数名字。	
	参数命名、顺序是否合理？	
	参数的个数是否太多？	
	是否使用类型和数目不确定的参数？	
	是否省略了函数返回值的类型？	
	函数名字与返回值类型在语义上是否冲突？	

重要	是否将正常值和错误标志混在一起返回？正常值应当用输出参数获得，而错误标志用 return 语句返回。	
重要	在函数体的“入口处”，是否用 assert 对参数的有效性进行检查？	
重要	使用滥用了 assert？ 例如混淆非法情况与错误情况，后者是必然存在的并且是一定要作出处理的。	
重要	return 语句是否返回指向“栈内存”的“指针”或者“引用”？	
	是否使用 const 提高函数的健壮性？const 可以强制保护函数的参数、返回值，甚至函数的定义体。“Use const whenever you need”	
	……	
内存管理		
重要性	审查项	结论
重要	用 malloc 或 new 申请内存之后，是否立即检查指针值是否为 NULL？（防止使用指针值为 NULL 的内存）	
重要	是否忘记为数组和动态内存赋初值？（防止将未被初始化的内存作为右值使用）	
重要	数组或指针的下标是否越界？	
重要	动态内存的申请与释放是否配对？（防止内存泄漏）	
重要	是否有效地处理了“内存耗尽”问题？	
重要	是否修改“指向常量的指针”的内容？	
重要	是否出现野指针？例如 （1）指针变量没有被初始化。 （2）用 free 或 delete 释放了内存之后，忘记将指针设置为 NULL。	
重要	是否将 malloc/free 和 new/delete 混淆使用？	
重要	malloc 语句是否正确无误？例如字节数是否正确？类型转换是否正确？	
重要	在创建与释放动态对象数组时，new/delete 的语句是否正确无误？	
	……	
C++ 函数的高级特性		
重要性	审查项	结论
	重载函数是否有二义性？	
重要	是否混淆了成员函数的重载、覆盖与隐藏？	
	运算符的重载是否符合制定的编程规范？	
	是否滥用内联函数？例如函数体内的代码比较长，函数体内出现循环。	

重要	是否用内联函数取代了宏代码？	
	……	
类的构造函数、析构函数和赋值函数		
重要性	审查项	结论
重要	是否违背编程规范而让 C++ 编译器自动为类产生四个缺省的函数：(1) 缺省的无参数构造函数；(2) 缺省的拷贝构造函数；(3) 缺省的析构函数；(4) 缺省的赋值函数。	
重要	构造函数中是否遗漏了某些初始化工作？	
重要	是否正确地使用构造函数的初始化表？	
重要	析构函数中是否遗漏了某些清除工作？	
	是否错写、错用了拷贝构造函数和赋值函数？	
重要	赋值函数一般分四个步骤：(1) 检查自赋值；(2) 释放原有内存资源；(3) 分配新的内存资源，并复制内容；(4) 返回 *this。是否遗漏了重要步骤？	
重要	是否正确地编写了派生类的构造函数、析构函数、赋值函数？注意事项： （1）派生类不可能继承基类的构造函数、析构函数、赋值函数。 （2）派生类的构造函数应在其初始化表里调用基类的构造函数。 （3）基类与派生类的析构函数应该为虚（即加 virtual 关键字）。 （4）在编写派生类的赋值函数时，注意不要忘记对基类的数据成员重新赋值。	
	……	
类的高级特性		
重要性	审查项	结论
重要	是否违背了继承和组合的规则？ （1）若在逻辑上 B 是 A 的“一种”，并且 A 的所有功能和属性对 B 而言都有意义，则允许 B 继承 A 的功能和属性。 （2）若在逻辑上 A 是 B 的“一部分”（a part of），则不允许 B 从 A 派生，而是要用 A 和其它东西组合出 B。	
	……	
其它常见问题		
重要性	审查项	结论
重要	数据类型问题：	

	（1）变量的数据类型有错误吗？ （2）存在不同数据类型的赋值吗？ （3）存在不同数据类型的比较吗？	
重要	变量值问题： （1）变量的初始化或缺省值有错误吗？ （2）变量发生上溢或下溢吗？ （3）变量的精度够吗？	
重要	逻辑判断问题： （1）由于精度原因导致比较无效吗？ （2）表达式中的优先级有误吗？ （3）逻辑判断结果颠倒吗？	
重要	循环问题： （1）循环终止条件不正确吗？ （2）无法正常终止（死循环）吗？ （3）错误地修改循环变量吗？ （4）存在误差累积吗？	
重要	错误处理问题： （1）忘记进行错误处理吗？ （2）错误处理程序块一直没有机会被运行？ （3）错误处理程序块本身就有毛病吗？如报告的错误与实际错误不一致，处理方式不正确等等。 （4）错误处理程序块是“马后炮”吗？如在被它被调用之前软件已经出错。	
重要	文件 I/O 问题： （1）对不存在的或者错误的文件进行操作吗？ （2）文件以不正确的方式打开吗？ （3）文件结束判断不正确吗？ （4）没有正确地关闭文件吗？	

附录 B ：C++/C 试题

本试题仅用于考查 C++/C 程序员的基本编程技能。内容限于 C++/C 常用语法，不涉及数据结构、算法以及深奥的语法。考试成绩能反映出考生的编程质量以及对 C++/C 的理解程度，但不能反映考生的智力和软件开发能力。

笔试时间 90 分钟。请考生认真答题，切勿轻视。

一、请填写 BOOL , float, 指针变量 与“零值”比较的 if 语句。（10 分）

提示：这里“零值”可以是 0, 0.0 , FALSE 或者“空指针”。例如 int 变量 n 与“零值”比较的 if 语句为：

```
if ( n == 0 )
if ( n != 0 )
```

以此类推。

请写出 BOOL flag 与“零值”比较的 if 语句：
请写出 float x 与“零值”比较的 if 语句：
请写出 char *p 与“零值”比较的 if 语句：

二、以下为 Windows NT 下的 32 位 C++程序，请计算 sizeof 的值（10 分）

<table>
<tr>
<td rowspan="2">char str[] = “Hello” ;
char *p = str ;
int n = 10;
请计算
sizeof (str) =

sizeof (p) =

sizeof (n) =</td>
<td>void Func (char str[100])
{
请计算
sizeof(str) =
}</td>
</tr>
<tr>
<td>void *p = malloc(100);
请计算
sizeof (p) =</td>
</tr>
</table>

三、简答题（25 分）

1、头文件中的 ifndef/define/endif 干什么用？

2、#include <filename.h> 和 #include “filename.h” 有什么区别？

3、const 有什么用途？（请至少说明两种）

4、在 C++ 程序中调用被 C 编译器编译后的函数，为什么要加 extern “C”声明？

5、请简述以下两个 for 循环的优缺点

```
// 第一个
for (i=0; i<N; i++)
{
  if (condition)
      DoSomething();
  else
      DoOtherthing();
}
```

```
// 第二个
if (condition)
{
    for (i=0; i<N; i++)
      DoSomething();
}
else
{
    for (i=0; i<N; i++)
      DoOtherthing();
}
```

第一个	第二个
优点：	优点：
缺点：	缺点：

四、有关内存的思考题（20 分）

<table>
<tr>
<td>

```
void GetMemory(char *p)
{
  p = (char *)malloc(100);
}
void Test(void)
{
  char *str = NULL;
  GetMemory(str);
  strcpy(str, "hello world");
  printf(str);
}
```

请问运行 Test 函数会有什么样的结果？
答：

</td>
<td>

```
char *GetMemory(void)
{
  char p[] = "hello world";
  return p;
}
void Test(void)
{
  char *str = NULL;
  str = GetMemory();
  printf(str);
}
```

请问运行 Test 函数会有什么样的结果？
答：

</td>
</tr>
<tr>
<td>

```
Void GetMemory2(char **p, int num)
{
  *p = (char *)malloc(num);
}
void Test(void)
{
    char *str = NULL;
    GetMemory(&str, 100);
    strcpy(str, "hello");
    printf(str);
}
```

请问运行 Test 函数会有什么样的结果？
答：

</td>
<td>

```
void Test(void)
{
    char *str = (char *) malloc(100);
    strcpy(str, “hello”);
    free(str);
    if(str != NULL)
    {
      strcpy(str, “world”);
      printf(str);
    }
}
```

请问运行 Test 函数会有什么样的结果？
答：

</td>
</tr>
</table>

五、编写 strcpy 函数（10 分）

已知 strcpy 函数的原型是

char *strcpy(char *strDest, const char *strSrc);

其中 strDest 是目的字符串，strSrc 是源字符串。

（1）不调用 C++/C 的字符串库函数，请编写函数 strcpy

（2）strcpy 能把 strSrc 的内容复制到 strDest，为什么还要 char * 类型的返回值？

六、编写类 String 的构造函数、析构函数和赋值函数（25 分）

已知类 String 的原型为：

```
class String
{
  public:
    String(const char *str = NULL);   // 普通构造函数
    String(const String &other);      // 拷贝构造函数
    ~ String(void);                      // 析构函数
    String & operate =(const String &other); // 赋值函数
  private:
    char    *m_data;               // 用于保存字符串
};
```

请编写 String 的上述 4 个函数。

附录 C ：C++/C 试题的答案与评分标准

一、请填写 BOOL , float, 指针变量 与“零值”比较的 if 语句。（10 分）

<table>
<tr><th colspan="2">请写出 BOOL flag 与“零值”比较的 if 语句。（3 分）</th></tr>
<tr><td>标准答案：
if (flag)
if (!flag)</td><td>如下写法均属不良风格，不得分。
if (flag == TRUE)
if (flag == 1)
if (flag == FALSE)
if (flag == 0)</td></tr>
<tr><th colspan="2">请写出 float x 与“零值”比较的 if 语句。（4 分）</th></tr>
<tr><td>标准答案示例：
const float EPSINON = 0.00001;
if ((x >= - EPSINON) && (x <= EPSINON)
不可将浮点变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”此类形式。</td><td>如下是错误的写法，不得分。
if (x == 0.0)
if (x != 0.0)</td></tr>
<tr><th colspan="2">请写出 char *p 与“零值”比较的 if 语句。（3 分）</th></tr>
<tr><td>标准答案：
if (p == NULL)
if (p != NULL)</td><td>如下写法均属不良风格，不得分。
if (p == 0)
if (p != 0)
if (p)
if (!)</td></tr>
</table>

二、以下为 Windows NT 下的 32 位 C++程序，请计算 sizeof 的值（10 分）

<table>
<tr><td rowspan="2">char str[] = “Hello” ;
char *p = str ;
int n = 10;
请计算
sizeof (str) = 6 （2 分）

sizeof (p) = 4 （2 分）

sizeof (n) = 4 （2 分）</td><td>void Func (char str[100])
{
请计算
sizeof(str) = 4 （2 分）
}</td></tr>
<tr><td>void *p = malloc(100);
请计算
sizeof (p) = 4 （2 分）</td></tr>
</table>

三、简答题（25 分）

1、头文件中的 ifndef/define/endif 干什么用？（5 分）
答：防止该头文件被重复引用。

2、#include <filename.h> 和 #include “filename.h” 有什么区别？（5 分）
答：对于#include <filename.h> ，编译器从标准库路径开始搜索 filename.h
对于#include “filename.h” ，编译器从用户的工作路径开始搜索 filename.h

3、const 有什么用途？（请至少说明两种）（5 分）
答：（1）可以定义 const 常量
（2）const 可以修饰函数的参数、返回值，甚至函数的定义体。被 const 修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。

4、在 C++ 程序中调用被 C 编译器编译后的函数，为什么要加 extern “C”？ （5 分）
答：C++语言支持函数重载，C 语言不支持函数重载。函数被 C++编译后在库中的名字与 C 语言的不同。假设某个函数的原型为： `void foo(int x, int y);`
该函数被 C 编译器编译后在库中的名字为 foo，而 C++编译器则会产生像 _foo_int_int 之类的名字。
C++提供了 C 连接交换指定符号 extern “ C ” 来解决名字匹配问题。

5、请简述以下两个 for 循环的优缺点（5 分）

``for (i=0; i<N; i++)`` `{` `if (condition)` `DoSomething();` `else` `DoOtherthing();` `}`	`if (condition)` `{` `for (i=0; i<N; i++)` `DoSomething();` `}` `else` `{` `for (i=0; i<N; i++)` `DoOtherthing();` `}`
优点：程序简洁 缺点：多执行了 N-1 次逻辑判断，并且打断了循环“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。	优点：循环的效率高 缺点：程序不简洁

四、有关内存的思考题（每小题 5 分，共 20 分）

<table>
<tr>
<td>

```
void GetMemory(char *p)
{
  p = (char *)malloc(100);
}
void Test(void)
{
  char *str = NULL;
  GetMemory(str);
  strcpy(str, "hello world");
  printf(str);
}
```

请问运行 Test 函数会有什么样的结果？
答：程序崩溃。
因为 GetMemory 并不能传递动态内存，Test 函数中的 str 一直都是 NULL。strcpy(str, "hello world");将使程序崩溃。

</td>
<td>

```
char *GetMemory(void)
{
  char p[] = "hello world";
  return p;
}
void Test(void)
{
  char *str = NULL;
  str = GetMemory();
  printf(str);
}
```

请问运行 Test 函数会有什么样的结果？
答：可能是乱码。
因为 GetMemory 返回的是指向“栈内存”的指针，该指针的地址不是 NULL，但其原现的内容已经被清除，新内容不可知。

</td>
</tr>
<tr>
<td>

```
void GetMemory2(char **p, int num)
{
  *p = (char *)malloc(num);
}
void Test(void)
{
    char *str = NULL;
    GetMemory(&str, 100);
    strcpy(str, "hello");
    printf(str);
}
```

请问运行 Test 函数会有什么样的结果？
答：
（1）能够输出 hello
（2）内存泄漏

</td>
<td>

```
void Test(void)
{
    char *str = (char *) malloc(100);
    strcpy(str, “hello”);
    free(str);
    if(str != NULL)
    {
      strcpy(str, “world”);
      printf(str);
    }
}
```

请问运行 Test 函数会有什么样的结果？
答：篡改动态内存区的内容，后果难以预料，非常危险。
因为 free(str);之后，str 成为野指针，if(str != NULL)语句不起作用。

</td>
</tr>
</table>

五、编写 strcpy 函数（10 分）

已知 strcpy 函数的原型是

char *strcpy(char *strDest, const char *strSrc);

其中 strDest 是目的字符串，strSrc 是源字符串。

（1）不调用 C++/C 的字符串库函数，请编写函数 strcpy

```
char *strcpy(char *strDest, const char *strSrc);
{
    assert((strDest!=NULL) && (strSrc !=NULL));   // 2分
    char *address = strDest;                      // 2分
    while( (*strDest++ = * strSrc++) != '\0' )    // 2分
       NULL ;
    return address ;                              // 2分
}
```

（2）strcpy 能把 strSrc 的内容复制到 strDest，为什么还要 char * 类型的返回值？

答：为了实现链式表达式。 // 2 分

例如 int length = strlen(strcpy(strDest, “hello world”));

六、编写类 String 的构造函数、析构函数和赋值函数（25 分）

已知类 String 的原型为：

```
class String
{
  public:
    String(const char *str = NULL);   // 普通构造函数
    String(const String &other);      // 拷贝构造函数
    ~ String(void);                   // 析构函数
    String & operate =(const String &other); // 赋值函数
  private:
    char    *m_data;                  // 用于保存字符串
};
```

请编写 String 的上述 4 个函数。

标准答案：

```
// String 的析构函数
String::~String(void)               // 3 分
{
    delete [] m_data;
    // 由于 m_data 是内部数据类型，也可以写成 delete m_data;
}
```

```
// String 的普通构造函数
String::String(const char *str)      // 6 分
{
    if(str==NULL)
    {
        m_data = new char[1];    // 若能加 NULL 判断则更好
        *m_data = '\0';
    }
    else
    {
        int length = strlen(str);
        m_data = new char[length+1];  // 若能加 NULL 判断则更好
        strcpy(m_data, str);
    }
}
// 拷贝构造函数
String::String(const String &other)   // 3 分
{
    int length = strlen(other.m_data);
    m_data = new char[length+1];      // 若能加 NULL 判断则更好
    strcpy(m_data, other.m_data);
}
// 赋值函数
String & String::operate =(const String &other)    // 13 分
{
    // (1) 检查自赋值                     // 4 分
    if(this == &other)
        return *this;

    // (2) 释放原有的内存资源            // 3 分
    delete [] m_data;

    // （3）分配新的内存资源，并复制内容 // 3 分
    int length = strlen(other.m_data);
    m_data = new char[length+1];         // 若能加 NULL 判断则更好
    strcpy(m_data, other.m_data);

    // （4）返回本对象的引用            // 3 分
    return *this;
}
```